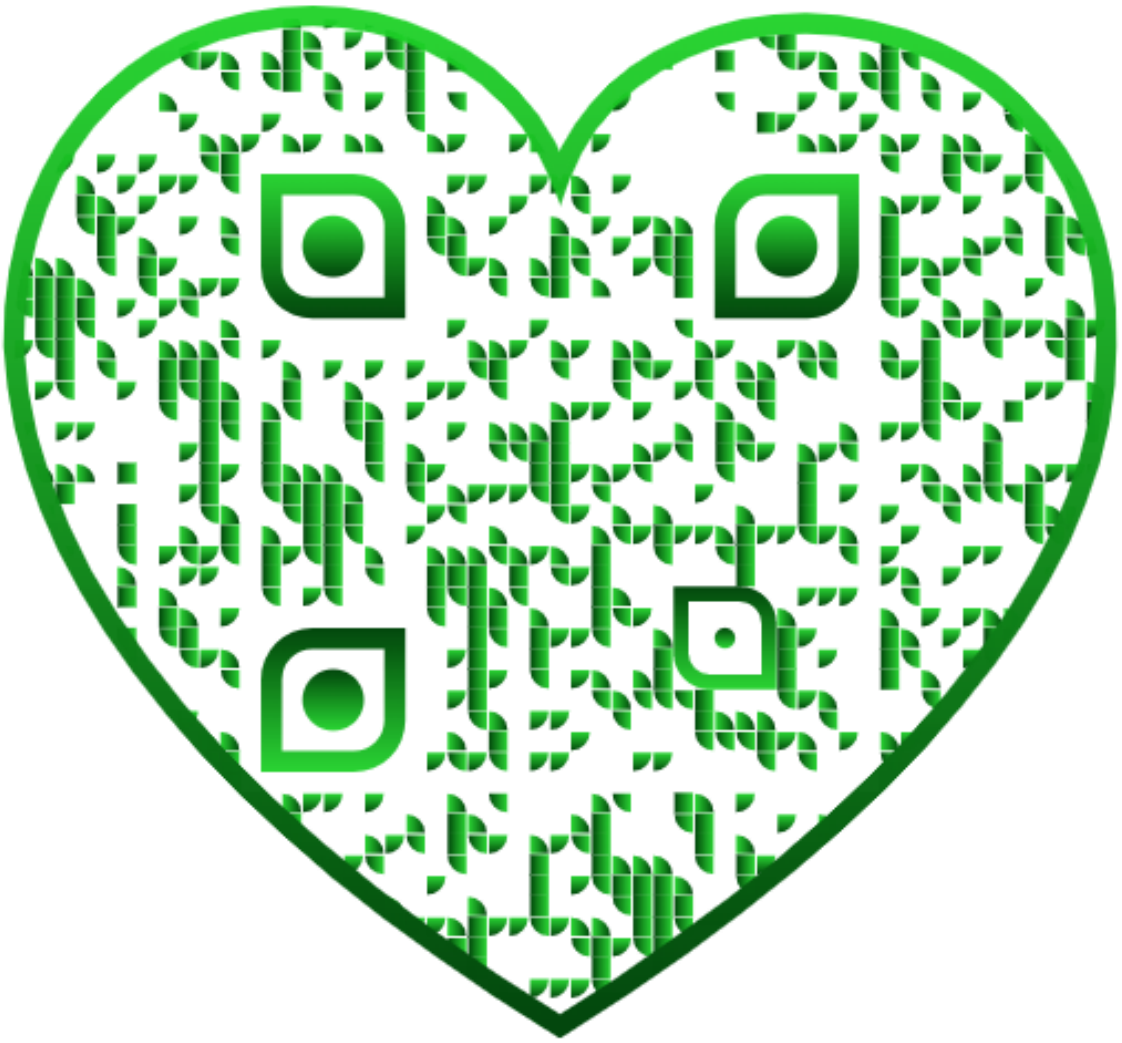
La première étape vers Python

Partie 1: Les fondamenteaux de Python 3

Edition: 1.0

Olivoy



La première Etape vers Python

Partie 1 : Les fondamentaux de python 3

Olivoy

Edition : 1.0

Copyright © 2019 Olivoy
Tous droits réservés

Tables des matières

- Introduction
- C’est quoi Python ?
- Élément nécessaire
- Python
  - Installation sous Windows
  - Installation sous Linux (Ubuntu)
- Édition du code source
- Votre premier programme
  - Commentaire
  - Variables
  - Noms des variables
  - Mots réservés
- Les opérateurs
  - Opérateurs arithmétiques
  - Opérateurs d’affectation
  - Opérateurs de comparaison
  - Opérateurs logiques
  - L’opérateur d’appartenance
- Les types de données
  - Numérique
    - Int (integer)
    - Float (nombre réel à virgule flottante)
    - Complex
  - Booléen
  - Les chaînes de caractères (String/str)

- Conversion
- L’indice d’une chaine de caractères
- Opérateur pour les chaines caractères :
- Formatage des chaînes
  - Ancien technique
  - Nouvelle technique
- Quelque Méthodes de l’objets string

Les listes

- Créer une liste
- L’indice (ou l'index) d’une liste
- Afficher un élément (item)
- Le slicing (extraire une partie d'une liste)
- Vérifier si un élément existe
- Listes imbriquées ou matrice
- Opérations sur les listes
- Fonctions utiles
- Modifier une liste
- Copier une liste
  - Une liste simple
  - Liste contenant des objets modifiables
- Méthodes de l’objet liste
  - La fonction del

Les tuples

- Créer un tuple
- L’indice, Le slicing et matrice
- Vérifier si un élément existe
- Opération sur les tuples
- Quelque fonction utile
- Méthodes de l’objets tuple

Dictionnaires

- Créer un dictionnaire
- Afficher un élément
- Ajouter un élément
- Modifier un élément
- Vérifier si une clé existe
- Méthodes de l’objet dictionnaire
- Copier un dictionnaire
  - Copie légère (shallow)
  - Copie complète (deepcopy)
- La fonction del

Ensembles (set)

Créer un ensemble(set)
Vérifier si un élément existe
Méthodes de l’objet set
Opération ensembliste
Ensembles (frozensets)
Créer un ensemble frozensets
Méthodes de l’objet frozensets
Comparaison entre les Collections
La fonction input()
Contrôle de flux
L'instruction if
L’instruction if avec and et or
La forme courte de l’instruction if
Les boucles
while
for
L’instruction else
Les boucles imbriquées
range()
Les instructions break et continue
Les fonctions
Appeler une fonction
Commentaire interne d'une fonction
Fonction sans paramètres
Fonction avec paramètres
Fonctions de rappel (callbacks)
Paramètre formel
return multiple
Ordre des arguments
Valeur par défaut des paramètres
Variable locale et globale

- Fonction lambda

Les modules

- Localisation des modules
- Lieu des modules créer
- Créer un module
- Utiliser un module
- Commentaire interne d'un module
- La fonction dir()

Les packages

- Créer packages
- Importer des packages
  - Méthode 1
  - Méthode 2
  - Méthode 3

Les exceptions

La programmation orientée objet

- Classe
- L'héritage de classe

Introduction

Salut, à la fin de ce livre vous serai capable de programmer avec Python, vous virez que c'est très facile, on va commencer étape par étape, et tout ça à l’aide des exemples et des explications bien objectif.

N’hésiter pas de m’informer s’il y a des manques, des remarques ou une partie n’est pas bien claire, je serai heureux de les ajouter dans les prochaines éditions, finalement ce livre et pour vous et je suis ici pour vous aider ;)

C’est quoi Python ?

Python c’est un langage de programmation créer en 1991 par Guido van Rossum, aujourd’hui c’est l’un des meilleurs langages vus qu’il est :

- Langage de programmation de haut niveau
- Facilité d'apprentissage
- Lisibilité des codes
- Entièrement gratuit
- Open source et multiplateforme (Windows, Mac, Linux, Raspberry Pi, etc).
- Portable et facile à intégrer
- Orienté objet
- De nombreuses bibliothèques basés sur Python sont disponibles

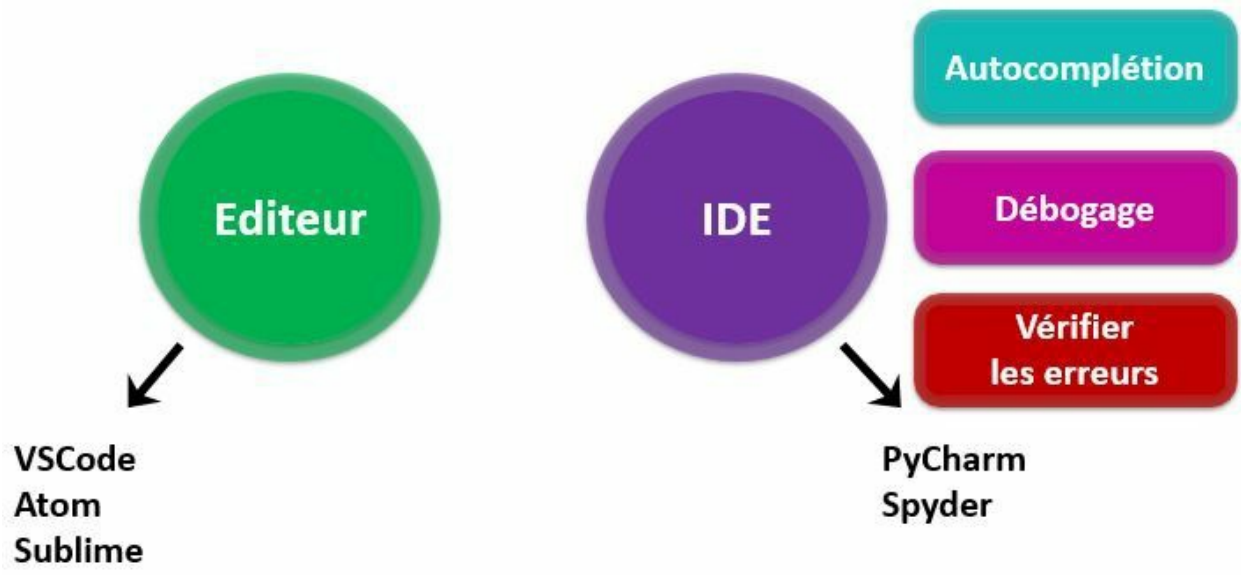
La version actuelle c’est python3 et c’est la version qu’on va l’apprendre dans ce livre, la version 2 sera abandonner à l’année 2020 donc c’est mieux d’apprendre la version Python3.

Élément nécessaire

Pour commencer notre programmation il nous faut deux éléments :

1. La dernière version de Python qui est disponible sur le site officiel https://www.python.org
2. Un outil où on va écrire notre code source en effet il y a deux types :

- Éditeur : tout simplement c'est un outil qui sert à écrire des textes en effet c'est un éditeur de texte, exemple :
  - Notepad
  - Vim
  - VsCode
- EDI (Environnement de Développement Intégré) : c'est l'inverse d'un éditeur de texte ; il permet la saisie semi-automatique aussi connue sous le nom d'IntelliSense, la détection les erreurs, déboguer et d'autre option, exemple :
  - PyCharm
  - Spyder

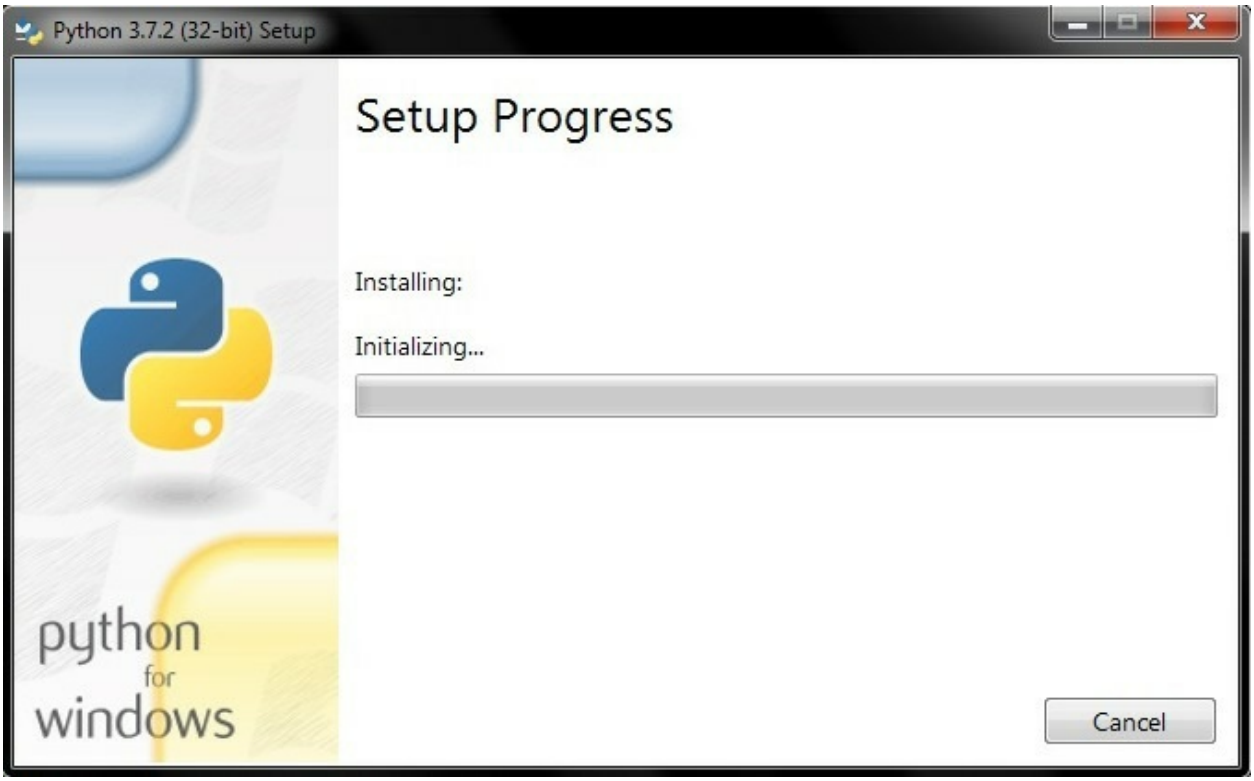


Python

Installation sous Windows

Après le téléchargement du fichier d'installation et l'exécuter la fenêtre suivante s'affiche :

1. Vous devez cocher la case « Add Python 3.7 to PATH » pour que vous pouvez lancer Python depuis votre ligne de commande Windows « cmd.exe » il suffit d'écrire python et valider par Entrée pour commencer le développement depuis votre terminal.
2. Maintenant vous cliquez sur « Install Now » pour continuer.


Python 3.7.2 (32-bit) Setup
Setup Progress
Installing:
Initializing...
python
for
windows
Cancel

Python 3.7.2 (32-bit) Setup
Setup was successful
Special thanks to Mark Hammond, without whose years of freely shared Windows expertise, Python for Windows would still be Python for DOS.
New to Python? Start with the online tutorial and documentation.
See what's new in this release.
python
for
windows
Close

Enfin un clic sur « Close » pour terminer l’installation.

Installation sous Linux (Ubuntu)

Python existe déjà sur Linux Mais avec la version 2. Donc pour l'installer taper la commande suivante dans votre terminal :

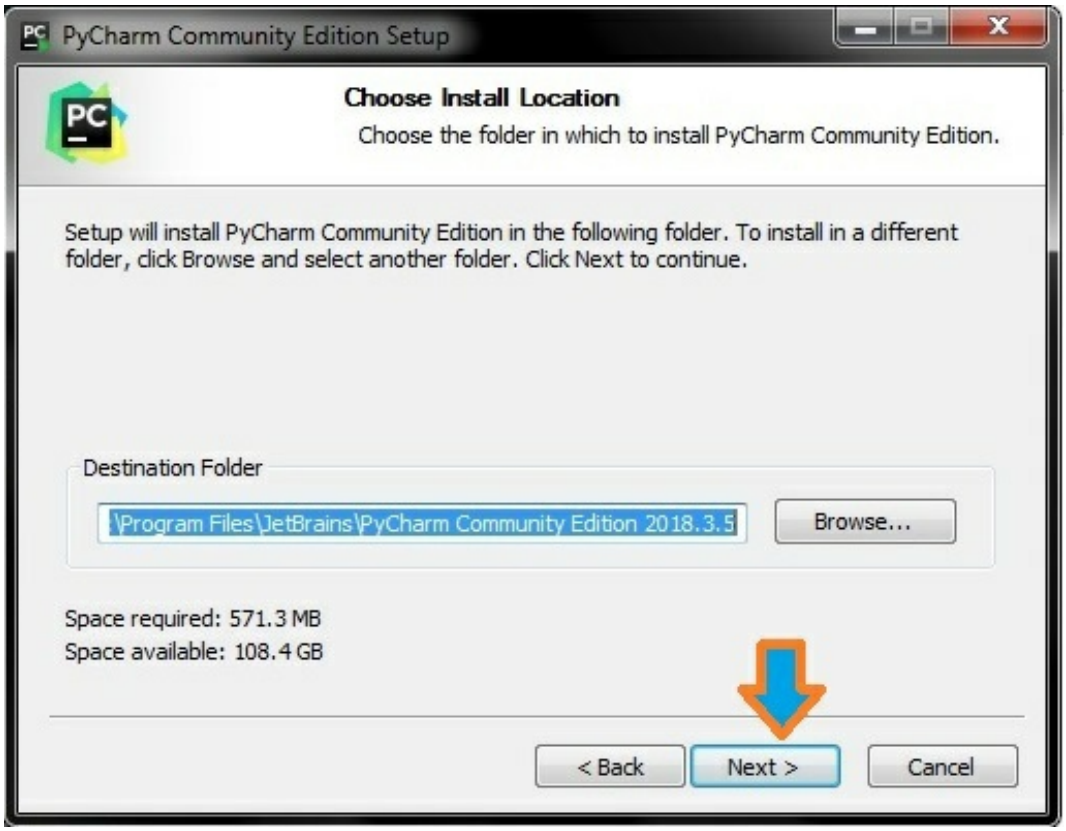
```
sudo apt-get install python3.5
```

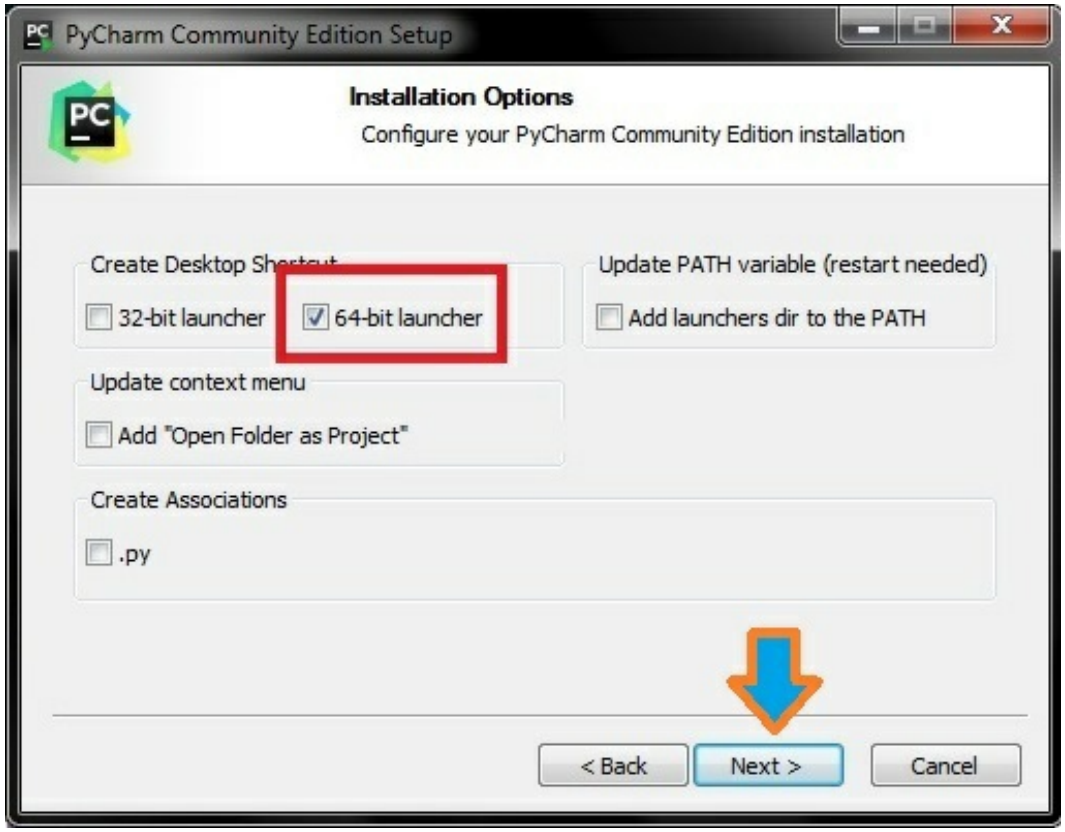
Lancer le terminal Linux et taper **python** mais attention vous devez ajouter le numéro de la version, si non la version 2 de python sera lance c'est pour ça écrire dans le terminal : **python3**

Édition du code source

Pour éditer notre code on va utiliser PyCharm, après le téléchargement du fichier d'installation de type gratuit «Community » depuis :

https://www.jetbrains.com/pycharm/download/#section=windows.

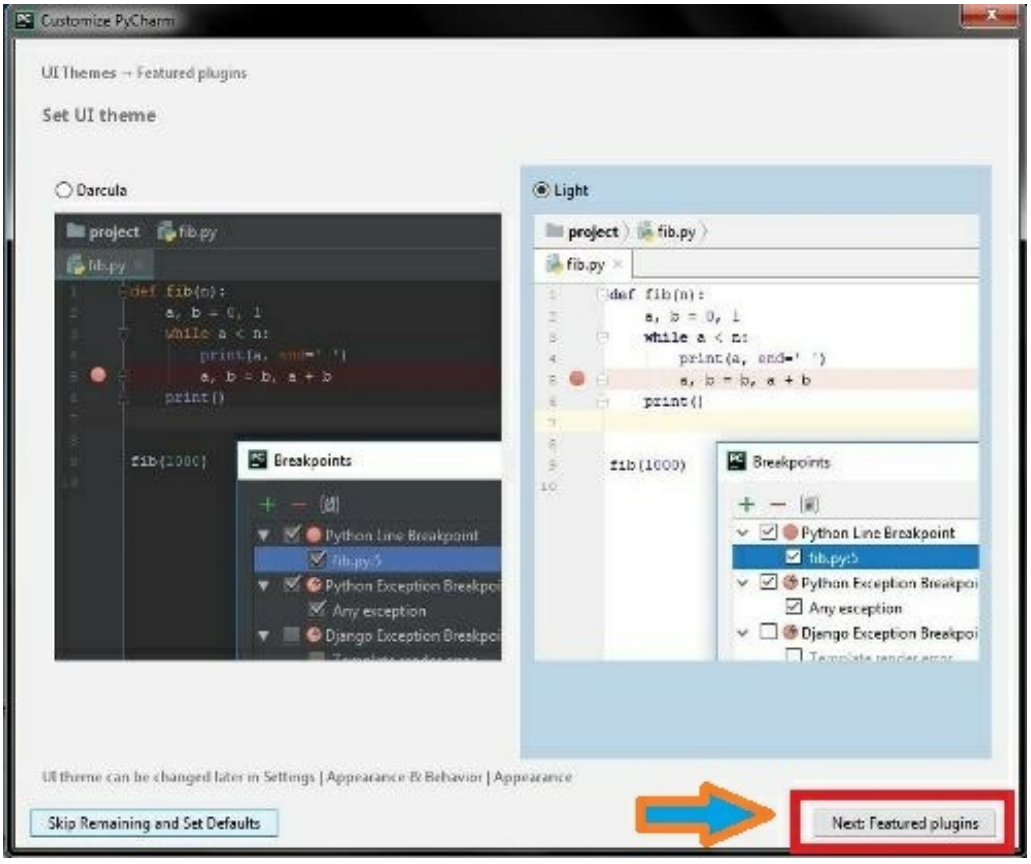

PyCharm Community Edition Setup
Choose Install Location
Choose the folder in which to install PyCharm Community Edition.
Setup will install PyCharm Community Edition in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.
Destination Folder
\Program Files\JetBrains\PyCharm Community Edition 2018.3.5
Browse...
Space required: 571.3 MB
Space available: 108.4 GB
< Back
Next >
Cancel


PyCharm Community Edition Setup
Installation Options
Configure your PyCharm Community Edition installation
Create Desktop Shortcut
32-bit launcher
64-bit launcher
Update PATH variable (restart needed)
Add launchers dir to the PATH
Update context menu
Add "Open Folder as Project"
Create Associations
.py
< Back
Next >
Cancel

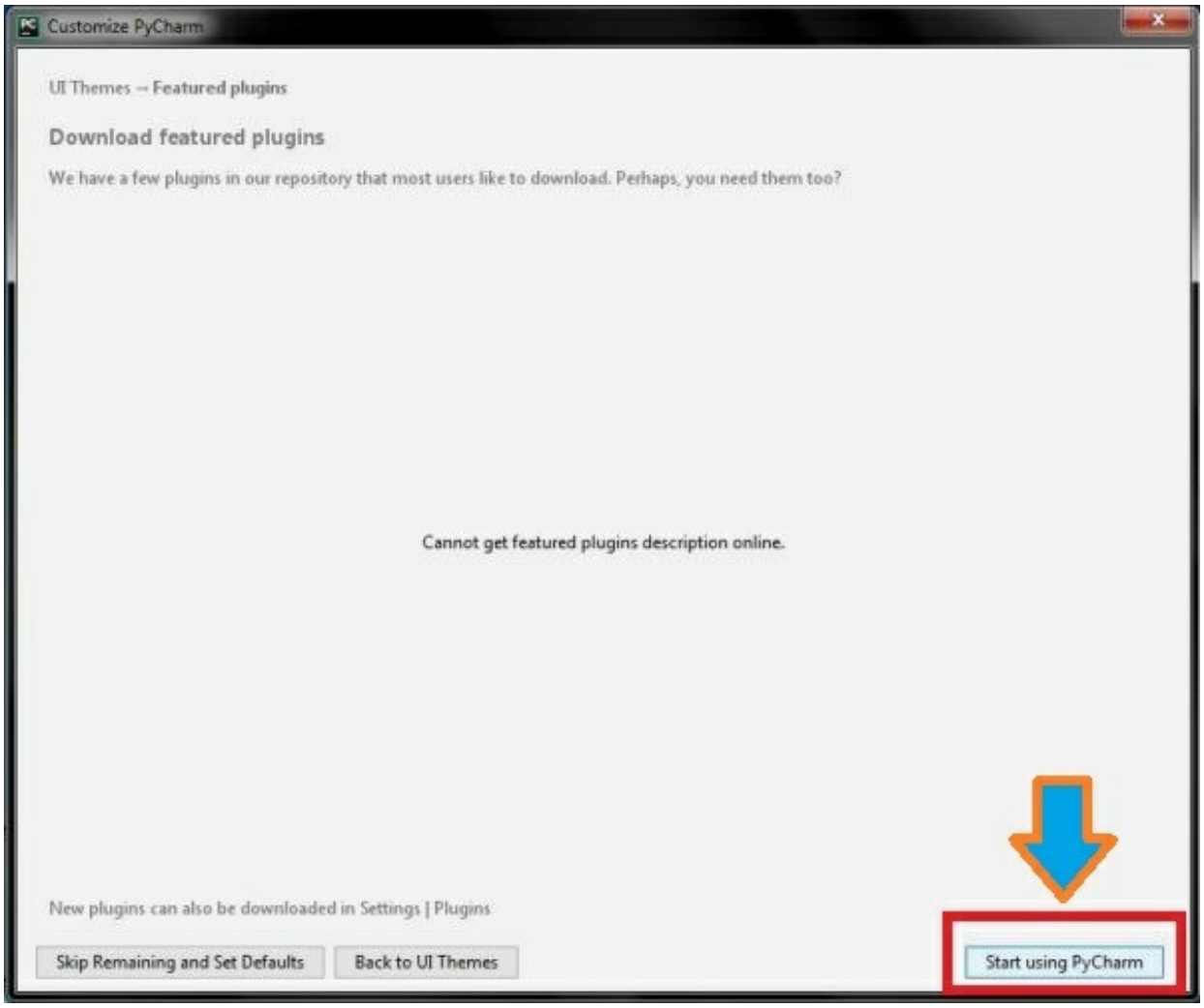
J’ai coché « 64-bit lancher » par-ce-que j’ai un système 64 bits, mais si vous utilisez 32 bits vous devez sélectionner l’autre case.

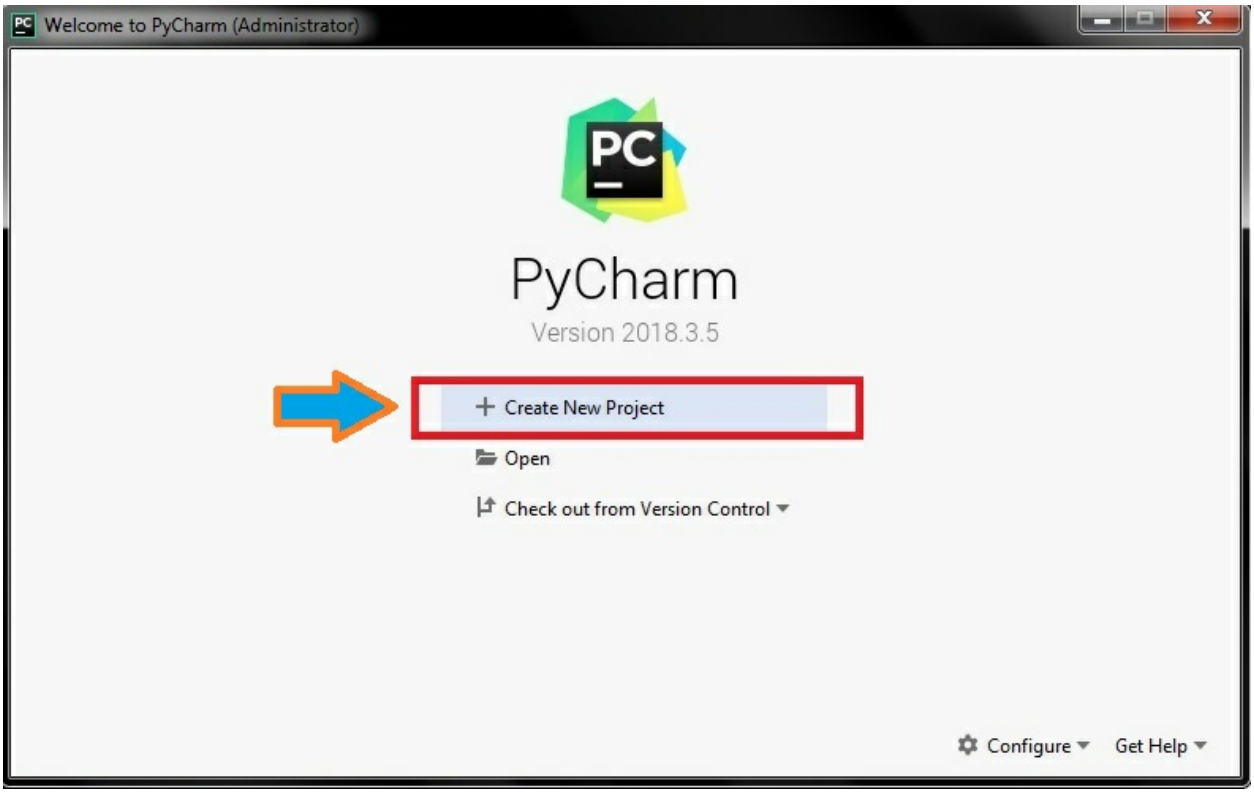
Après l’installation l’icône suivante s’affiche dans votre bureau :

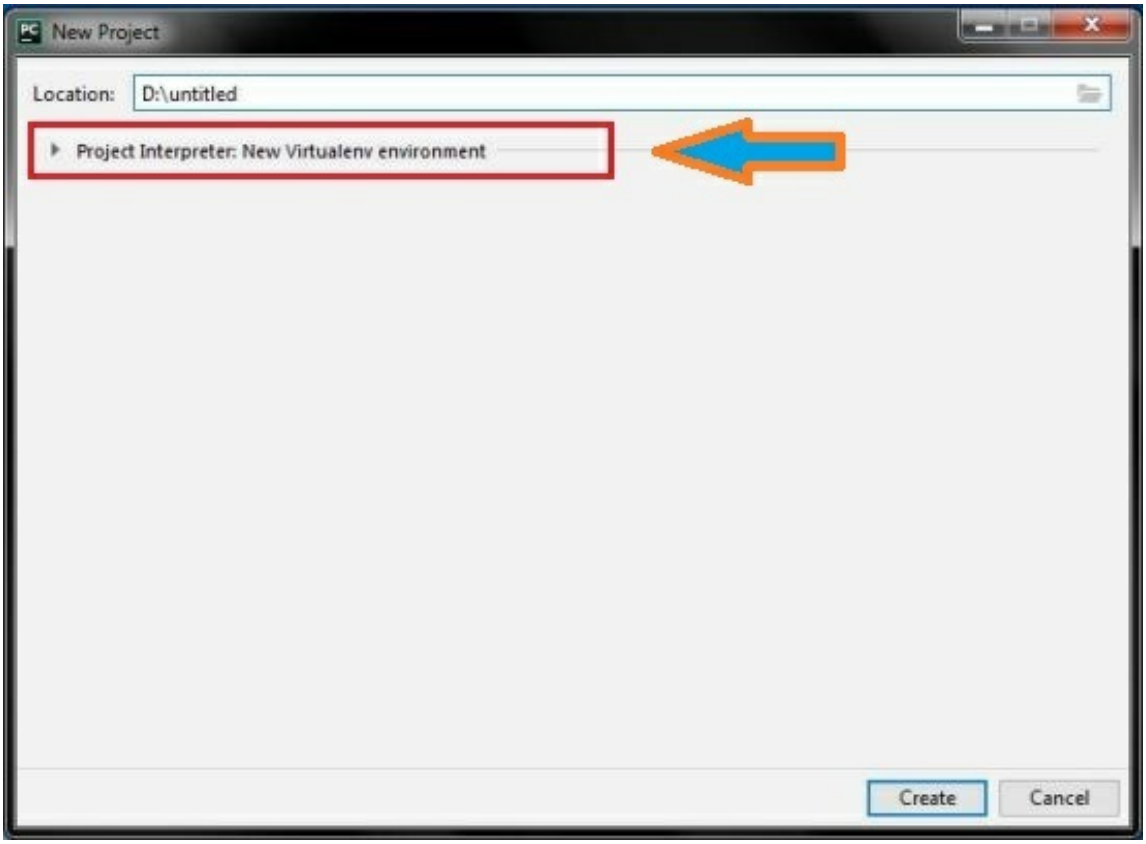
Pour la première exécution de PyCharm la fenêtre suivante s’affiche :

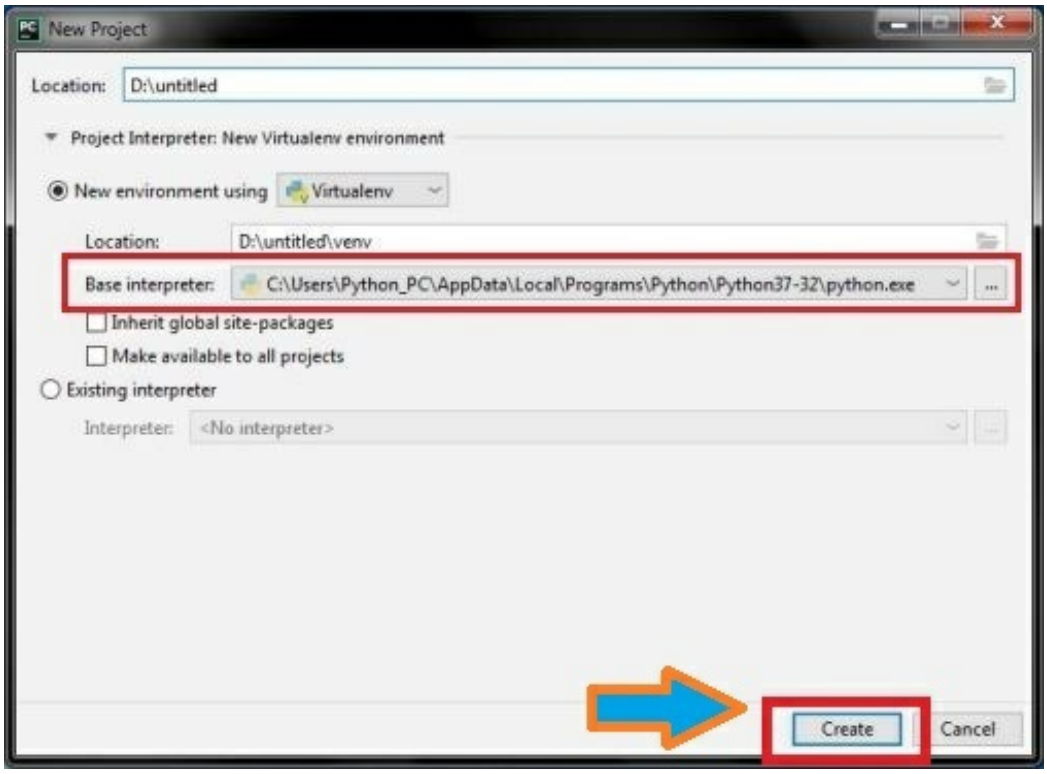


Si vous travaillez beaucoup dans la nuit je te conseil de choisir le thème **Dracula** c’est bien pour vos yeux.

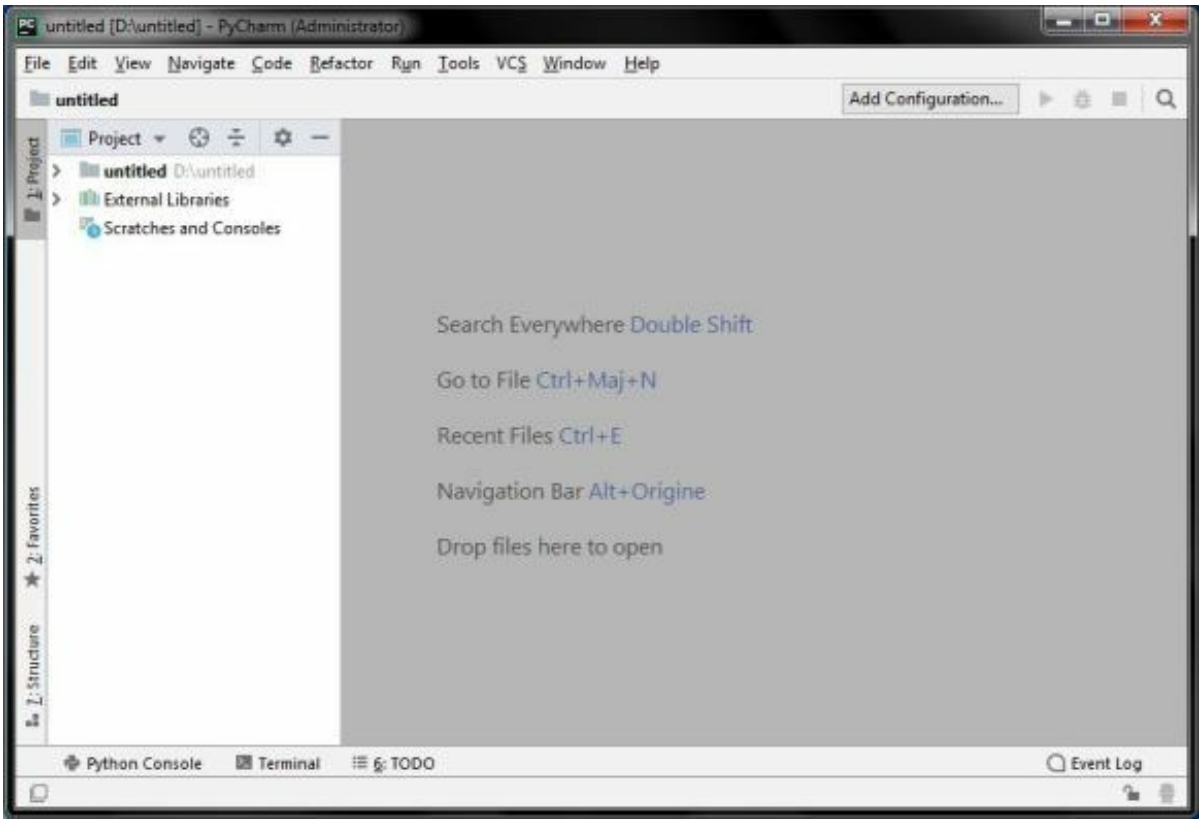

Customize PyCharm
UI Themes → Featured plugins
Download featured plugins
We have a few plugins in our repository that most users like to download. Perhaps, you need them too?
Cannot get featured plugins description online.
New plugins can also be downloaded in Settings | Plugins
Skip Remaining and Set Defaults
Back to UI Themes
Start using PyCharm


Welcome to PyCharm (Administrator)
PyCharm
Version 2018.3.5
Create New Project
Open
Check out from Version Control
Configure
Get Help

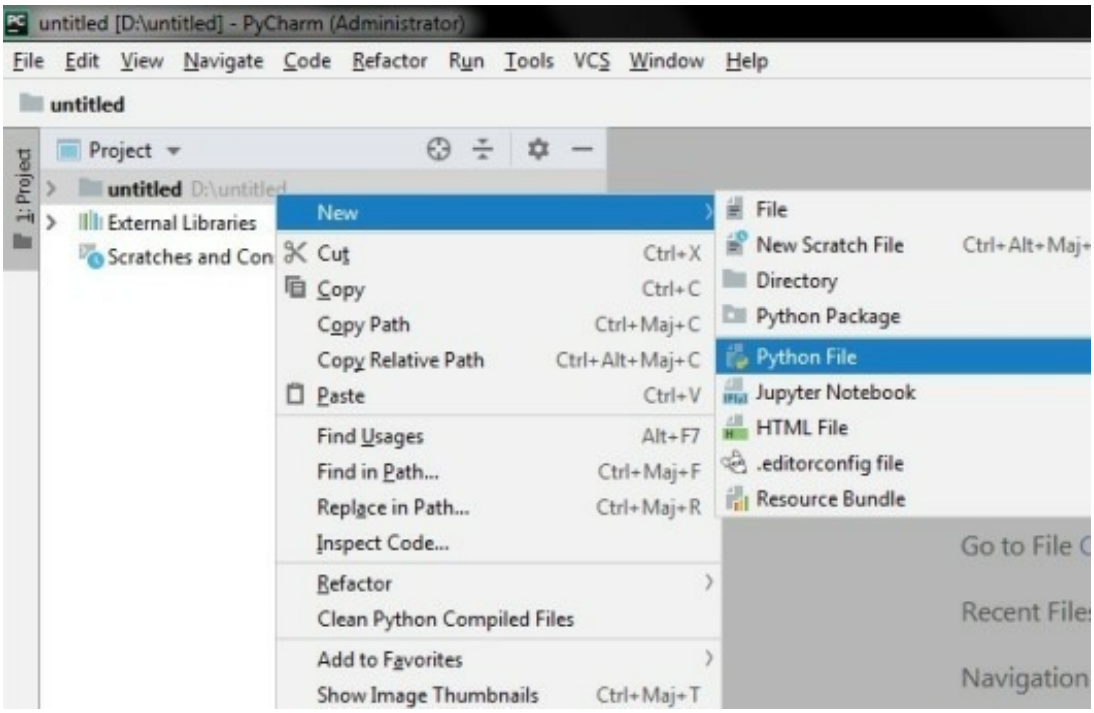

New Project
Location:
D:\untitled
Project Interpreter: New Virtualenv environment
Create
Cancel



Le chemin par défaut de l'interpréteur Python mais si vous installez Python dans un répertoire personnalisé vous devez l'ajouter.

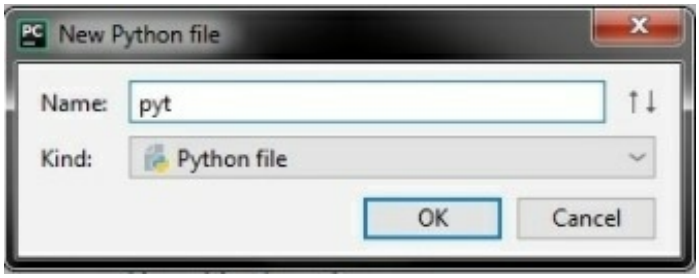


Enfin voilà l'environnement de travail.

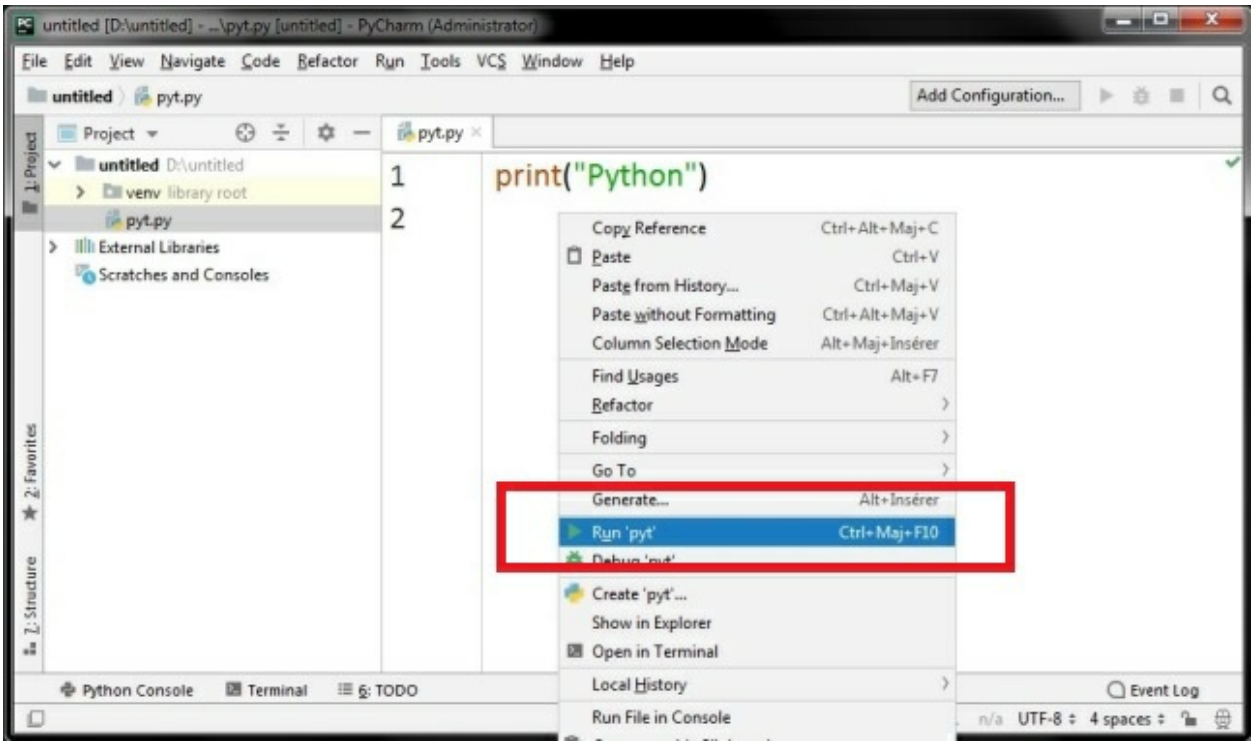


Pour ajouter un nouveau fichier Python pour commencer le codage vous devez faire un clic droit sur le répertoire du projet **untiled** suivi par **new** et **Python File**

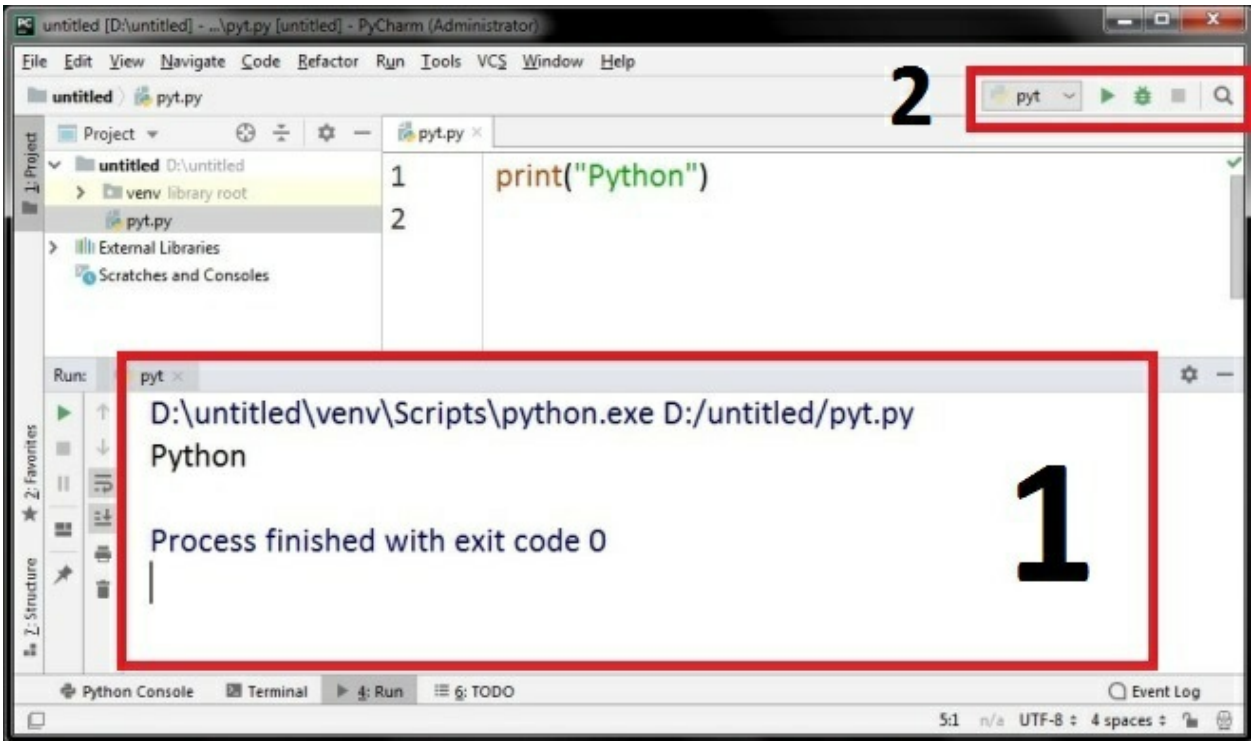
Après la fenêtre s'affiche pour que vous entrez le nom du fichier.



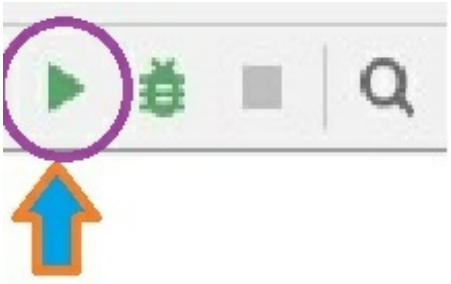
Après vous validez par « OK »



Après l'ajout d'un code, un clic droit sur la page de développement et sélectionner « Run »



Le résultat de l'exécution s'affiche dans la partie **1**, remarque que la partie **2** devient active, donc la prochaine exécution vous pouvez le faire depuis cette partie **2** on cliquons sur le triangle vert.



Votre premier programme

Notre première ligne du code va afficher à l'écran un message traditionnel dans le domaine de programmation, c'est la fameuse phrase "Hello world !", mais nous on va l'écrire en français ;)

```
print("Bonjour tout le monde !")
```

Résultat :

```
D:\Python_cours\venv\Scripts\python.exe D:/Python_cours/code_Python.py
Bonjour tout le monde !

Process finished with exit code 0
```

Comme vous remarquez pour afficher un message on doit utiliser la fonction **print()** et comme c'est une fonction il faut toujours utiliser les double cote **()** et à l'intérieur on écrire notre message avec deux "" ou avec une seul ‘’

Commentaire

Une chose plus important qu'il faut l'apprendre en premier c'est savoir comment ajouter un commentaire à notre code, par-ce-que c'est une chose important d'écrire des remarques dans notre code c'est vraiment une chose qui va vous aidez à organiser votre code et rapidement résoudre les problèmes.

Tous simplement mettre le caractère spécial dièse # devant le commentaire.

Exemple :

```
# Première commentaire
print("Bonjour tout le monde !") # Deuxième commentaire
```

Si vous voulez écrire votre commentaire sur plusieurs lignes vous pouvez utiliser un Docstring ; qui commence par trois apostrophes ou guillemets suivez par votre commentaire et terminé par d'autre trois apostrophes ou guillemets.

Exemple :

```
"""
Commentaire sur
Plusieurs
Ligne

"""
```

Variables

Nous pouvons dire qu'une variable rassemble à une boite dans laquelle on peut stocker une ou plusieurs données autrement dit c'est une espace réserver temporairement dans le mémoire de votre ordinateur où on va stocker nos valeurs.

Dans les autres langages de programmation, les variables doivent être déclarer ; c'est-à-dire définir le type en premier, par exemple dans la langage C pour créer une variable **x** qui est numérique avec une valeur initiale **1**, il faut faire (**int x = 1;**). Mais dans Python il suffit d'écrire seulement le nom de votre variable avec une initialisation de la valeur, après Python créée automatiquement la variable avec le type qui correspond au mieux à la valeur fournie, par exemple :

```
x = 1
print(x)
print(type(x))
```

Résultat :

```
1
<class 'int'>
```

- La première **print** affiche la valeur de **x**.
- La deuxième **print** affiche le résultat de la fonction **type(x)**.

Note :

La fonction **type()** affiche le type d'une variable, dans notre exemple le résultat de cette fonction c'est <class 'int'> c'est-à-dire que la variable x est numérique(integer).

Exemple :

```
x = "Bonjour tout le monde !"
print(x)
y = type(x)
print(y)
```

Résultat :

```
Bonjour tout le monde !
<class 'str'>
```

Note :

- Comme vous remarquez je n'ai pas affiché directement le résultat de la fonction **type(x)**, on peut stocker le résultat dans une variable nommée y et après afficher la valeur de y via la fonction **print()**
- <class 'str'> ça veut dire type texte(string)

Noms des variables

Avant tout vous devez savoir que Python est sensible à la casse c’est-à-dire qu’il fait la différence entre le minuscule et la majuscule, par exemple les variables Nom, nom et NOM sont déférentes.

- Le nom d'une variable peut être écrite par des lettres minuscules, majuscules, des nombres ou la caractère souligné (_)
- Le nom d'une variable doit débuter par des lettres ou par le caractère souligné (_) et non par un nombre
- Le nom d'une variable ne doive pas contient d’espace, remplacer-le par le caractère souligné (_)
- Il est interdit d’utiliser les mots « réservé » par Python

Mots réservés

Je ne vais pas faire comme les autres mais je vais te donner une commande qui affiche les mots réservés, donc s'il y aura un nouveau mot réservé vous pouvez servir de cette commande a tout le temps.

Exemple :

```
import keyword

print(keyword.kwlist)
```

Résultat :

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await',
'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except',
'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is',
'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try',
'while', 'with', 'yield']
```

Note : dans les prochains chapitres on va bien expliquer l'utilité du mots import.

Les opérateurs

Les opérateurs sont des symboles ou des mots réservés utilisé pour effectuer des opérations sur les variables.

Opérateurs arithmétiques

Comme exemple on a : **x** avec la valeur 10 et **y** avec la valeur 6

Symbole	Description	Exemple	Résultat
+	Addition	x + y (10+6)	16
-	Soustraction	x – y (10-6)	4
*****	Multiplication	x * y (10*6)	60
/	Division réelle	x / y (10/6)	1.6666666666666667
//	Division entière	x // y (10//6)	1
%	Modulo : reste de la **division entière**	x % y (10%6)	4
******	Exponentiel	x ** y (10**6)	1000000

Opérateurs d’affectation

C’est donner une valeur à une variable créée à l’aide du symbole = et vous pouvez changer la valeur initiale quand vous voulez (réaffectation).

Exemple :

```
a = 1
b = 3 * 2

x = y = 1  # Affectations multiples
z, w = 2, 6  # Affectations parallèles
# z reçoit 2 et w reçoit 6
```

Symbole	Exemple	Equivalent
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
//=	x //= 5	x = x // 5
%=	x %= 5	x = x % 5
**=	x **= 5	x = x ** 5

Opérateurs de comparaison

Le résultat des opérateurs de comparaison donnera une valeur booléenne True ou False; True si la comparaison et bien correcte ou False si le contraire.

Comme exemple on a : **x** avec la valeur 10 et **y** avec la valeur 6

Symbole	Description	Exemple	Résultat
==	Egal	x == y (10==6)	False
!=	Différent	x != y (10 !=6)	True
>	Strictement supérieur	x > y (10 >6)	True
<	Strictement inférieur	x < y (10 <6)	False
>=	Supérieur ou égal	x >= y (10 >=6)	True
<=	Inférieur ou égal	x <= y (10 <=6)	False

Opérateurs logiques

Tout comme les opérateurs de comparaison, le résultat des opérateurs logiques donnera une valeur booléenne True ou False.

Comme exemple on a : **x** avec la valeur 10

Symbole	Description	Exemple	Résultat
and	Retourne True si les deux conditions sont correct, mais si l'un des deux sont incorrect le résultat sera False.	x < 20 and x > 5	True
		x < 20 and x == 15	False
or	Retourne True si l'un des deux condition sont correct mais si les deux et incorrect le résultat sera False.	x < 50 or x == 20	True
		x < 5 or x == 6	False
not	Retourne le contraire d'une résulta.	not(x == 10)	False

L’opérateur d’appartenance

Vérifier si un élément donné appartient à un objet.

Comme exemple on a : **x** avec la valeur 10

Symbole	Exemple	Résultat
in	x in (2, 5, 15, 10)	True
not in	x not in ("a", "b", 2)	True

Les types de données

Dans ce cours on va voir 6 types :

- Numérique
- Booléen
- Les chaînes de caractères
- Les liste
- Les tuple
- Les dictionnaires
- set

Numérique

Il y a trois catégories de type numérique (int, float et complex).

Exemple :

```
x = 5  # Nombre entier
# (en anglais integer ou int)

y = 1.5  # Nombre réel ou Nombre à virgule flottante
# (en anglais Floating point number ou float)

z = 3j  # Nombre complexe
# (en anglais complex)
```

Vous pouvez utiliser la fonction type(), elle vous indique le type d'une variable, voilà comment l'utiliser :

Exemple :

```
x = 5
print("Le type de la variable 'x' :", type(x))

y = 1.5
print("Le type de la variable 'y' :", type(y))

z = 3j
# Création de la variable "z_type"
# Stocke le résultat de la fonction "type" dans z_type
z_type = type(z)
print("Le type de la variable 'z' :", z_type)
```

Résultat :

```
Le type de la variable 'x' : <class 'int'>
Le type de la variable 'y' : <class 'float'>
Le type de la variable 'z' : <class 'complex'>
```

Int (integer)

C'est un nombre entier positif ou négative avec une taille dépond de la capacité mémoire de votre ordinateur.

Float (nombre réel à virgule flottante)

C’est un nombre positif ou négative avec virgule.

Vous pouvez écrire un nombre float sous forme scientifique en utilisant un exposant de 10 (e ou E suivie par un nombre), par exemple e3 c’est 10 exponentiel 3 (10**3) c’est-à-dire 1000

Exemple :

```
x = 12e3
y = 12E2
z = 1.5e5

print("La valeur de x :" ,x)
print("La valeur de y :", y)
print("La valeur de z :", z)
print("Le type de x, y, z :",type(x), type(y), type(z))
```

Résultat :

```
La valeur de x : 12000.0
La valeur de y : 1200.0
La valeur de z : 150000.0
Le type de x, y, z : <class 'float'> <class 'float'> <class 'float'>
```

Complex

C’est un nombre avec une partie imaginaire qui est toujours indiquer par « **j** » et une partie réelle.

C’est un type de base pour les nombres complexes **C** créée sous forme algébrique **a + ib**, où a et b sont des nombres réels et **i** c'est un nombre imaginaire tel que $\mathbf{i^2 = -1}$

Note :

- Une nombre complexe s'écrit toujours comme suivant : **a + bj**
- **a** et **b** des variables de type float
- Toujours attacher un nombre avec **j** si non Python va le considérer comme variable
- Pas d'espace entre **j** et son nombre attacher

Exemple :

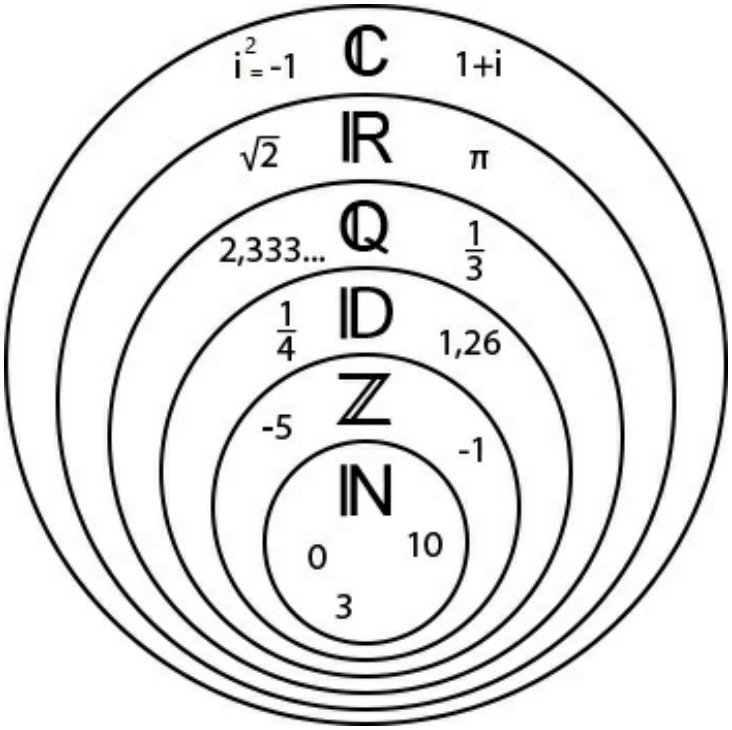
```
x = 1 - 5j
y = 3 + 2j

print("La somme de x et y :", x + y)
print("La partie réelle de y :", y.real)
print("La partie imaginaire de y :", y.imag)
```

Résultat :

```
La somme de x et y : (4-3j)
La partie réelle de y : 3.0
La partie imaginaire de y : 2.0
```

Petit rappel sur les ensembles mathématiques :



Booléen

C’est un type de donnée qui prend seulement deux valeur : vrais ou faux en Python c’est True et False (la première lettre est majuscule).

Exemple :

```
x = True
y = False
print("Le type de x :", type(x))
print("Le type de y :", type(y))
```

Résultat :

```
Le type de x : <class 'bool'>
Le type de y : <class 'bool'>
```

Les chaînes de caractères (String/str)

C’est une suite de caractères écrit soit entre deux apostrophes ('), soit entre des guillemets (") ou par des triples quotes (apostrophes ou guillemets) (''' ou """).

Note : vous pouvez utiliser les triples quotes pour écrire votre commentaire sur plusieurs lignes.

Si votre chaine de caractères est entourée par des guillemets (") et que vous souhaitez utiliser le caractère (") à l’intérieur de votre chaine il y aura deux solution pour l'écrire, si non Python va détectera une erreur :

- Soit vous utilisez une séquence d'échappement antislash (\) suivie par le caractère (")
- Soit vous utilisez une apostrophe (') pour délimitée votre chaine

Voilà les principaux séquences d’échappement :

\\	Antislash
\'	Apostrophe
\"	Guillemet
n	Saut de ligne
t	Tabulation

Exemple :

```
print("Le fichier est dans le dossier D:\\Dossier1\\")
```

Résultat :

```
Le fichier est dans le dossier  D:\Dossier1\
```

Exemple :

```
# on n'a pas utilisé une séquence d'échappement antislash (\)
# Par ce que notre texte est entourée par des guillemets
print ("Voici une apostrophe '")
```

Résultat :

```
Voici une apostrophe '
```

Exemple :

```
print ("Voici des Guillemets \"")
```

Résultat :

```
Voici des Guillemets "
```

Exemple :

```
print ("Voici \nUn saut de ligne")
```

Résultat :

```
Voici
Un saut de ligne
```

Exemple :

```
print("\tVoici une tabulation")
```

Résultat :

```
        Voici une tabulation
```

Conversion

Vous pouvez convertir un type vers un autre via les fonctions suivant : int, float, complex, bool et str.

Exemple avant l'utilisation de la conversion :

```
1  x = 10
2  # on va concaténer "chaine1" avec x
3  concat = "chaine1" + x
4  print(concat)
```

Note : comme vous remarquer PyCharm marque et souligne **x** !!!! alors si vous mettre votre curseur sur **x** vous obtiendrais.

Note : pour voir plus des détails sur l'erreur, vous cliquez sur more… ou appuyer sur (Ctrl+F1).

Résultat :

```
Traceback (most recent call last):
  File "D:/Python_cours/code_Python.py", line 3, in <module>
    concat = "chaine1" + x
TypeError: can only concatenate str (not "int") to str
```

Note : après l'exécution de notre code, un message indique qu'il y a une erreur dans la ligne 3, seulement le type « str » qui peut ce concatène, alors la solution c'est convertir x vers le type chaine de caractère (String/str).

Exemple :

```
x = 10
# on va concaténer "chaine1" avec x
concat = "chaine1" + str(x)
print(concat)
```

Résultat :

```
chaine110
```

Exemple :

```
nbr1 = 20
nbr2 = "100"
somme = nbr1 + int(nbr2)
print("La somme de nbr1 et nbr2 égale :", somme)
```

Résultat :

```
La somme de nbr1 et nbr2 égale : 120
```

Exemple :

```
x = 0
y = bool(x)
print("La valeur de y :", y)
```

Résultat :

```
La valeur de y : False
```

Note : **0**, **""** et **None** sont convertie en False, mais le reste c'est True (un espace est considéré comme un caractère c'est pour ça " " est converti en True).

L'indice d'une chaine de caractères

L'indice (ou l'index) est utilisé pour déterminer la position d'un caractère ou une partie d'une chaine de caractères, on vira par la suite comment utiliser cet index mais pour l'instant il faut savoir comment déterminer l'index, voyons l'exemple suivant :

Chaine de caractères	**B**	**o**	**n**	**j**	**o**	**u**	**r**
Indice positif	0	1	2	3	4	5	6
Indice négatif	-7	-6	-5	-4	-3	-2	-1

Opérateur pour les chaines caractères :

+	Concatène deux chaines de caractères.
*****	Multiplier la chaine de caractères par un nombre.
[]	Slice :Donne un caractère d'un index donné.
[:]	Range Slice :Donne les caractères d'une plage donnée [n: m] avec **n** inclus et **m** exclus.
Len()	Retourne le nombre des caractères d'une chaine (Longueur).

in	Retourne True si un caractère existe dans une chaine donnée.
not in	Retourne True si un caractère n'existe pas dans une chaine donnée

Exemple :

```
chaine1 = "Bonjour"
chaine2 = "tout le monde"

# Concaténation de deux chaines de caractères
print("Concaténation :", chaine1 + " " + chaine2)

# Multiplication d'une chaine de caractères
print("Multiplication :", chaine1 * 3)

# Vérifier si un caractère existe
verifier1 = "B" in chaine1
print("'B' existe dans 'chaine1' :", verifier1)

# Vérifier si un caractère n'existe pas
verifier2 = "t" not in chaine1
print("'t' n'existe pas dans 'chaine1' :", verifier2)
```

Résultat :

```
Concaténation : Bonjour tout le monde
Multiplication : BonjourBonjourBonjour
'B' existe dans 'chaine1' : True
't' n'existe pas dans 'chaine1' : True
```

Exemple :

```
chaine1 = "Bonjour"
chaine2 = "tout le monde"

# Le nombre des caractères d'une chaine
print("Longueur de 'chaine1' :", len(chaine1))

# Slice
print("Slice [0] :", chaine1[0])

# Range slice
rg1 = chaine1[3:7]
print("Range slice [3:7] :", rg1)

# Les 4 premiers caractères
# C'est-à-dire 0,1,2,3
# N'oublier pas que 4 n'est pas inclus
rg2 = chaine1[:4]
print("Les 4 premiers caractères [:4] :", rg2)

# Tout ce qui suit les 2 premiers caractères
# C'est-à-dire que 0 et 1 n'est pas afficher
# L'affichage commence par l'index 2 jusqu'à la fin
rg3 = chaine1[2:]
```

Résultat :

```
Longueur de 'chaine1' : 7
Slice [0] : B
Range slice [3:7] : jour
Les 4 premiers caractères [:4] : Bonj
Tout ce qui suit les 3 premiers caractères [3:] : jour
```

Formatage des chaînes

Vous pouvez changer une partie de votre chaine de caractères et insérer des variables sans utilisé la concaténation (+) et la conversion des variables numérique, on peut réaliser cette formatage avec la méthode format(), voyons des exemples pour bien comprendre l'utilisation de formatage.

Ancien technique

Cette méthode utilise le symbole % pour formater la chaine, mais c'est à toi de définir le type de formatage, par exemple pour afficher un nombre entier vous devez utiliser %d

%s	chaine de caractère (utilise la fonction str() pour la conversion)
%d	nombre entier
%g	floatte

Exemple :

```
nom = "user1"
age = 30

print("Votre nom est %s et tu es âgé %d ans" % (nom, age))
```

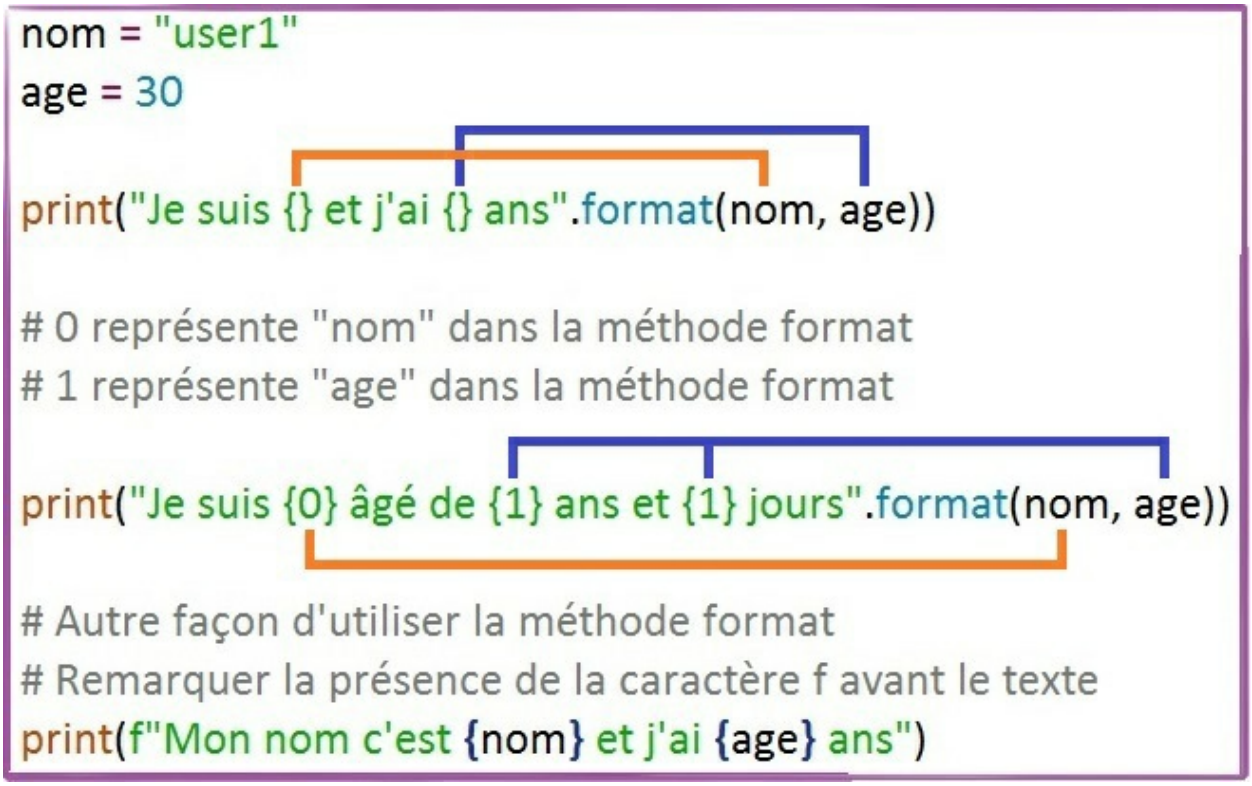
Résultat :

```
Votre nom est user1 et tu es âgé 30 ans
```

Nouvelle technique

La nouvelle façon consiste d'utiliser la méthode format() avec des arguments qui contient les valeurs à insérer dans votre chaine de caractères.

Exemple :



```
nom = "user1"
age = 30

print("Je suis {} et j'ai {} ans".format(nom, age))

# 0 représente "nom" dans la méthode format
# 1 représente "age" dans la méthode format

print("Je suis {0} âgé de {1} ans et {1} jours".format(nom, age))

# Autre façon d'utiliser la méthode format
# Remarquer la présence de la caractère f avant le texte
print(f"Mon nom c'est {nom} et j'ai {age} ans")
```

Résultat :

```
Je suis user1 et j'ai 30 ans
Je suis user1 âgé de 30 ans et 30 jours
Mon nom c'est user1 et j'ai 30 ans
```

Quelque Méthodes de l’objets string

Méthode	Description
upper()	Convertit la chaîne de caractère en majuscules.
lower()	Convertit la chaîne de caractère en minuscules.
title()	Convertit la première lettre de tous les mot d'une chaîne de caractère en majuscule.
capitalize()	Convertit la première lettre de la première mot d'une chaîne de caractère en majuscule.
strip()	Supprime l'espace au début et à la fin de la chaîne de caractère.
find()	L'index de la première caractère du mot recherché, mais s'il n’existe pas le résultat sera -1.
replace("a ", "b ")	Remplace tous les caractères "a" par le nouveau caractère "b".

Exemple :

```
msg = "bonjour TOUT le monde"

print("Convertit msg en majuscule :", msg.upper())
print("Convertit msg en minuscule :", msg.lower())
print("Tous les premières lettres en majuscule :", msg.title())
print("La première lettre en majuscule :", msg.capitalize())
print("Supprime l'espace au début et à la fin :", msg.strip())
print("L'index du mot recherché :", msg.find("jo"))
print("Remplace 'o' par 'a' :", msg.replace("o", "a"))
```

Résultat :

```
Convertit msg en majuscule : BONJOUR TOUT LE MONDE
Convertit msg en minuscule : bonjour tout le monde
Tous les premières lettres en majuscule : Bonjour Tout Le Monde
La première lettre en majuscule : Bonjour tout le monde
Supprime l'espace au début et à la fin : bonjour TOUT le monde
L'index du mot recherché : 3
Remplace 'o' par 'a' : banjaur TOUT le mande
```

N'oubliez pas que Python est sensible à la casse, la lettre "o" n'est pas comme la lettre "O"

Les résultats de ces méthodes ne modifier pas la chaine origine, mais si vous voulez garder les modifications vous devez affecter les résultats de ces méthodes à une nouvelle variable.

Exemple :

```python
msg = "bonjour TOUT le monde"

nouveau_msg = msg.upper()
print(f"Message origine : {msg}")
print(f"Le nouveau message : {nouveau_msg}")
```

Résultat :

```
Message origine : bonjour TOUT le monde
Le nouveau message : BONJOUR TOUT LE MONDE
```

Pour voir tous les autres méthodes utilisée la commande suivante :

```
print(help(str))
```

Les listes

Les listes sont des Objets qui contient un ensemble d'éléments(valeurs) de n'importe quel type, séparés par des virgules et délimiter par des crochets.

Créer une liste

Exemple :

```
# Création d'une liste vide
ma_liste1 = list()

# Création d'une liste qui contient des objets
ma_liste2 = ['a', 'Lundi', 'b', 1]

print("Le type de ma_liste1 :", type(ma_liste1))
print(f"Le contenu de ma_liste2 : {ma_liste2}")
```

Résultat :

```
Le type de ma_liste1 : <class 'list'>
Le contenu de ma_liste2 : ['a', 'Lundi', 'b', 1]
```

L'indice (ou l'index) d'une liste

C'est le même principe utiliser sur les chaines de caractères, mais pour les listes l'indice est donné pour chaque objet de la liste et non pour chaque caractère.

Liste	**'a'**	**'Lundi'**	**'b'**	**1**
Indice positif	0	1	2	3
Indice négatif	-4	-3	-2	-1

Afficher un élément (item)

Pour afficher un élément d'une liste, il suffit d’écrire liste[n] où n est l'indice de l’élément que vous voulez l'afficher.

Exemple :

```
ma_liste = ['a', 'Lundi', 'b', 1]

print(f"Le premier élément : {ma_liste[0]}")
print(f"Le deuxième élément : {ma_liste[1]}")
print(f"Le dernier élément : {ma_liste[-1]}")
```

Résultat :

```
Le premier élément : a
Le deuxième élément : Lundi
Le dernier élément : 1
```

Le slicing (extraire une partie d'une liste)

Vous pouvez afficher une sous-liste via une intervalle d’indice [m,n] (**m** inclut mais **n** exclu) .

Exemple :

```
ma_liste = ['a', 'Lundi', 'b', 1]

print(f"Les deux premiers éléments : {ma_liste[0:2]}")
print(f"Tout ce qui suit le premier élément : {ma_liste[1:]}")
```

Résultat :

```
Les deux premiers éléments : ['a', 'Lundi']
Tout ce qui suit le premier élément : ['Lundi', 'b', 1]
```

Vérifier si un élément existe

Exemple :

```
ma_liste = ['a', 'Lundi', 'b', 1]

recherche1 = "Lundi" in ma_liste
recherche2 = "Mardi" in ma_liste
print(f"Lundi se trouve dans ma_liste ? {recherche1}")
print(f"Mardi se trouve dans ma_liste ? {recherche2}")
```

Résultat :

```
Lundi se trouve dans ma_liste ? True
Mardi se trouve dans ma_liste ? False
```

Listes imbriquées ou matrice

Il est possible de créer une liste à l'intérieur d'une autre liste, pour le faire c'est très simple :

Exemple :

```
ma_liste = ['Lundi', 'Mardi', ['Janvier', 'Février', 'Mars'],
'Mercredi']

element = ma_liste[0]
print(f"Le premier élément : {element}")
```

Résultat :

```
Le premier élément : Lundi
```

Pour accéder à l'élément « Janvier » vous devez tout d'abord déterminer l'indice de la liste intérieure ; dans notre cas c'est 2, après vous ajoutez l'indice de l'élément de la deuxième liste qui est 0.

Liste	**'Lundi'**	**'Mardi'**	**['Janvier', 'Février', 'Mars']**	**'Mercredi'**
Indice positif	0	1	2	3
Indice négatif	-4	-3	-2	-1

Exemple :

```
ma_liste = ['Lundi', 'Mardi', ['Janvier', 'février', 'Mars'],
'Mercredi']

element1 = ma_liste[2]
print(f"La liste intérieure : {element1}")

# 2 est l' indice de la liste qui est à l'intérieur de la liste
principale.
# 0 est l' indice de "Janvier" dans la liste intérieure.
```

Résultat :

```
La liste intérieure : ['Janvier', 'février', 'Mars']
Le premier élément de la deuxième liste : Janvier
```

Exemple :

```
ma_liste1 = [1, 2, 3]
ma_liste2 = [4, 5, 6]
ma_liste3 = [7, 8, 9]

matrice_lst = [ma_liste1, ma_liste2, ma_liste3]

print(f"Tous les listes : {matrice _lst}")
print(f"La première petite liste : {matrice_lst[0]}")
```

Résultat :

```
Tous les listes : [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
La première petite liste : [1, 2, 3]
Le premier élément de la deuxième petite liste : 4
```

Opérations sur les listes

Exemple :

```
liste1 = [1, 2]
liste2 = [3, 4, 5]

# La concaténation +
conc_liste = liste1 + liste2
print(f"La concaténation de liste1 et liste2 : {conc_liste}")

# La multiplication *
mult_liste = liste1 * 3
print(f"La multiplication de liste1 par 3 : {mult_liste}")
```

Résultat :

```
La concaténation de liste1 et liste2 : [1, 2, 3, 4, 5]
La multiplication de liste1 par 3 : [1, 2, 1, 2, 1, 2]
```

Fonctions utiles

Exemple :

```
liste1 = [4, 5, 2, 9, 10, 15]

# La longueur d'une liste
len_liste = len(liste1)
print(f"La longueur de liste1 : {len_liste}")

# La valeur maximum d'une liste
mx_liste = max(liste1)
print(f"La valeur maximum de liste1 : {mx_liste}")

# La valeur minimum d'une liste
mn_liste = min(liste1)
print(f"La valeur minimum de liste1 : {mn_liste}")
```

Résultat :

```
La longueur de liste1 : 6
La valeur maximum de liste1 : 15
La valeur minimum de liste1 : 2
```

Modifier une liste

Exemple :

```
liste1 = [4, 5, 2, 9, 10, 15]

print(f"La valeur de l' indice [0] : {liste1[0]}")

# Modifier la valeur de l'indice 0
liste1[0] = 90
print(f"La nouvelle valeur de l'indice [0] : {liste1[0]}")
```

Résultat :

```
La valeur de l' indice [0] : 4
La nouvelle valeur de l'indice [0] : 90
```

Copier une liste

La première chose que tout le monde pense à faire, c'est affecter directement une liste créer à la nouvelle liste par exemple : liste2=liste1, mais cette technique ne marche pas sur les objets, on dit que liste2 est un alias du nom liste1, donc si vous modifiez l'un des deux listes automatiquement vous modifier l'autre.

Donc on peut dire que cette méthode affecte seulement le référence vers la zone mémoire où il est stockée notre liste.

Pour bien comprendre on va utiliser l'identifiant (id) de chaque objet, cette identifiant nous permet de savoir où il est stocké en mémoire.

Exemple :

```
liste1 = [1, 2, 3]
liste2 = liste1

# L'identifiant pour les deux listes
print("L'identifiant de liste1 :", id(liste1))
print("L'identifiant de liste2 :", id(liste2))

# Modification de liste2
liste2[0]=50
print(f"liste1 après la modification de liste2 : {liste1}")
print(f"liste2 après la modification : {liste2}")

# L'identifiant pour les deux listes après la modification
print("L'identifiant de liste1 après modification :", id(liste1))
print("L'identifiant de liste2 après modification :", id(liste2))
```

Résultat :

```
L'identifiant de liste1 : 1852880
L'identifiant de liste2 : 1852880
liste1 après la modification de liste2 : [50, 2, 3]
liste2 après la modification : [50, 2, 3]
L'identifiant de liste1 après modification : 1852880
L'identifiant de liste2 après modification : 1852880
```

Une liste simple

Si votre liste contient des variables et non des objets modifiables comme une autre liste par exemple, alors vous pouvez soit utiliser la méthode copy() ou la fonction list().

Ces techniques créent des copies légères de type shallow.

Exemple :

```
liste1 = [1, 2, 3]
liste2 = liste1.copy()
print(f"Le contenu de liste1 : {liste1}")
print(f"Le contenu de liste2 : {liste2}")
# Modification de liste2
liste2[0] = 50
print(f"liste1 après la modification de liste2 : {liste1 }")
print(f"Le contenu de liste2 après la modification : {liste2}")
```

Résultat :

```
Le contenue de liste1 : [1, 2, 3]
Le contenue de liste2 : [1, 2, 3]
liste1 après la modification de liste2 : [1, 2, 3]
Le contenue de liste2 après la modification : [50, 2, 3]
```

Exemple :

```
liste1 = [4, 5, 6]
liste2 = list(liste1)
print(f"Le contenu de liste1 : {liste1}")
print(f"Le contenu de liste2 : {liste2}")
# Modification de liste2
liste2[0] = 10
print(f"liste1 après la modification de liste2 : {liste1}")
print(f"Le contenu de liste2 après la modification : {liste2}")
```

Résultat :

```
Le contenue de liste1 : [4, 5, 6]
Le contenue de liste1 : [4, 5, 6]
liste1 après la modification de liste2 : [4, 5, 6]
Le contenue de liste1 après la modification : [10, 5, 6]
```

Liste contenant des objets modifiables

Pour faire une copie d'une liste qui contient une autre liste, vous devez faire une copie complète (deepcopy).

Pour réaliser cette copie vous devez utiliser la fonction deepcopy du module copy.

Mais avant, on va faire une copie légère pour une liste contenant une autre liste.

Exemple :

```
liste1 = [1, 2, 3, [10, 20]]
liste2 = liste1.copy()
print(f"Le contenu de liste1 : {liste1}")
print(f"Le contenu de liste2 : {liste2}")
# Modification de la liste qui est à l'intérieur de liste2
liste2[3][0] = 50
print(f"liste1 après la modification de liste2 : {liste1}")
print(f"Le contenu de liste2 après la modification : {liste2}")
```

Résultat :

```
Le contenu de liste1 : [1, 2, 3, [10, 20]]
Le contenu de liste2 : [1, 2, 3, [10, 20]]
liste1 après la modification de liste2 : [1, 2, 3, [50, 20]]
Le contenu de liste2 après la modification : [1, 2, 3, [50, 20]]
```

Note : Comme vous remarquez, la modification de liste1 affect l'autre liste.

Exemple :

```
import copy

liste1 = [1, 2, 3, [10, 20]]
liste2 = copy.deepcopy(liste1)
print(f"Le contenu de liste1 : {liste1}")
print(f"Le contenu de liste2 : {liste2}")
# Modification de la liste qui est à l'intérieur de liste2
liste2[3][0] = 50
print(f"liste1 après la modification de liste2 : {liste1}")
print(f"Le contenu de liste2 après la modification : {liste2}")
```

Résultat :

```
Le contenu de liste1 : [1, 2, 3, [10, 20]]
Le contenu de liste2 : [1, 2, 3, [10, 20]]
liste1 après la modification de liste2 : [1, 2, 3, [10, 20]]
Le contenu de liste2 après la modification : [1, 2, 3, [50, 20]]
```

Note : avec la fonction deepcopy(), la copie c'est bien passé même si on modifie liste2 l'autre liste reste sans modification.

Méthodes de l’objet liste

Méthode	Description
append(x)	Ajouter l'élément x à la fin de liste.
insert(n,x)	Insérer un élément x à une position n.
remove(x)	Supprimer l'élément x d'une liste. Si aucun élément ne correspond pas à la valeur fournie, une erreur est retournée.
clear()	Supprimer tous les éléments d'une liste.
extend(liste2)	Ajouter les éléments de liste2 en fin d'une autre liste.
index(x)	Retourne l’indice du premier élément de la valeur x. Si aucun élément ne correspond pas à la valeur x, une erreur s’est produite.
count(x)	Compter le nombre d’occurrences de l’élément x.
sort()	Trier les éléments d'une liste.
reverse()	Inverse l'ordre des éléments d'une liste.
pop(n)	Supprimer et afficher l'élément de l'indice n. Si vous utilisez pop() sans indice alors c'est l’élément dernier qui sera traité.
copy()	Faire une copie légère de type shallow.

Exemple :

```
liste1 = [1, 2, 3]

print(f"Le contenu de liste1 : {liste1}")
liste1.append('Lundi')
print(f"liste1 après l'utilisation de 'append' : {liste1}")
```

Résultat :

```
Le contenu de liste1 : [1, 2, 3]
liste1 après l'utilisation de 'append' : [1, 2, 3, 'Lundi']
```

Exemple :

```
liste1 = [1, 2, 3]

print(f"Le contenu de liste1 : {liste1}")
liste1.insert(0,'Lundi')
print(f"liste1 après l'utilisation de 'insert' : {liste1}")
```

Résultat :

```
Le contenu de liste1 : [1, 2, 3]
liste1 après l'utilisation de 'insert' : ['Lundi', 1, 2, 3]
```

Exemple :

```python
liste1 = [10, 20, 30]

print(f"Le contenu de liste1 : {liste1}")
liste1.remove(10)
print(f"liste1 après l'utilisation de 'remove' : {liste1}")
```

Résultat :

```
Le contenu de liste1 : [10, 20, 30]
liste1 après l'utilisation de 'remove' : [20, 30]
```

Exemple :

```python
liste1 = [10, 20, 30]

print(f"Le contenu de liste1 : {liste1}")
liste1.clear()
print(f"liste1 après l'utilisation de 'clear' : {liste1}")
```

Résultat :

```
Le contenu de liste1 : [10, 20, 30]
liste1 après l'utilisation de 'clear' : []
```

Exemple :

```python
liste1 = [1, 2, 3]
liste2 = [4, 5, 6]

print(f"Le contenu de liste1 : {liste1}")
liste1.extend(liste2)
print(f"liste1 après l'utilisation de 'extend' : {liste1}")
```

Résultat :

```
Le contenu de liste1 : [1, 2, 3]
liste1 après l'utilisation de 'extend' : [1, 2, 3, 4, 5, 6]
```

Exemple :

```
liste1 = ['a', 'Lundi', 'b', 1]

recherche1 = liste1.index("Lundi")
print(f"L'indice du mot 'Lundi' dans liste1 : {recherche1}")
```

Résultat :

```
L'indice du mot 'Lundi' dans liste1 : 1
```

Exemple :

```
liste1 = ['a', 'Lundi', 'b', 'Lundi']

cmp = liste1.count('Lundi')
print(f"Le nombre d' occurrences du mot 'Lundi' : {cmp}")
```

Résultat :

```
Le nombre d' occurrences du mot 'Lundi' : 2
```

Exemple :

```
liste1 = [50, 3, 5, -1,0]

print(f"liste1 avant l'utilisation de 'sort' : {liste1}")
liste1.sort()
print(f"liste1 après l'utilisation de 'sort' : {liste1}")
```

Résultat :

```
liste1 avant l'utilisation de 'sort' : [50, 3, 5, -1, 0]
liste1 après l'utilisation de 'sort' : [-1, 0, 3, 5, 50]
```

Exemple :

```
liste1 = [50, 3, 5, -1,0]

print(f"liste1 avant l'utilisation de 'reverse' : {liste1}")
liste1.reverse()
print(f"liste1 après l'utilisation de 'reverse' : {liste1}")
```

Résultat :

```
liste1 avant l'utilisation de 'reverse' : [50, 3, 5, -1, 0]
liste1 après l'utilisation de 'reverse' : [0, -1, 5, 3, 50]
```

Exemple :

```
liste1 = [1, 2, 3, 4]

print("L'élément qui sera supprimer :", liste1.pop(0))
print(f"liste1 après l'utilisation de pop(0) : {liste1}")
print("L'élément qui sera supprimer :", liste1.pop())
print(f"liste1 après l'utilisation de pop() : {liste1}")
```

Résultat :

```
L'élément qui sera supprimer : 1
liste1 après l'utilisation de pop(0) : [2, 3, 4]
L'élément qui sera supprimer : 4
liste1 après l'utilisation de pop() : [2, 3]
```

La fonction del

Au lieu de supprimer un élément par sa valeur, vous pouvez utiliser l'indice pour la suppression. Vous pouvez aussi utiliser la fonction del pour supprimer des tranches ou vider complètement la liste comme la méthode clear().

Exemple :

```
liste1 = [10, 20, 30, 40, 50, 60]
print(f"liste1 avant l'utilisation de la fonction del : {liste1}")

# Supprimer l'élément de l'indice [0]=10
del liste1[0]
print(f"Après la suppression de l'indice 0 : {liste1}")

# Supprimer la tranche [0:3]=[20,30,40]
del liste1[0:3]
print(f"Après la suppression de la tranche [0:3] : {liste1}")

# Supprimer tous les éléments
del liste1[:]
print(f"Après la suppression complète : {liste1}")
```

Résultat :

```
liste1 avant l'utilisation de la fonction del : [10, 20, 30, 40, 50, 60]
Après la suppression de l'indice 0 : [20, 30, 40, 50, 60]
Après la suppression de la tranche [0:3] : [50, 60]
Après la suppression complète : []
```

Les tuples

C’est comme une liste, mais il y a deux grande différence :

1. Un tuple n'est pas modifiable
2. L’ensemble est entouré par des parenthèses

Créer un tuple

Exemple :

```
# Création d'un tuple vide
tup1 = tuple()

# Création d'un tuple qui contient des objets
tup2 = ('a', 'Lundi', 'b', 1)

print("Le type de tup1 :", type(tup1))
print(f"Le contenu de tup2 : {tup2}")
```

Résultat :

```
Le type de tup1 : <class 'tuple'>
Le contenu de tup2 : ('a', 'Lundi', 'b', 1)
```

L’indice, Le slicing et matrice

Le même principe des listes s’applique aussi sur les tuples sauf que vous ne pouvez pas modifier ou ajouter des éléments à un tuple déjà créé, mais si tu as un tuple qui contient une liste dans ce cas vous pouvez modifier seulement la liste qui est à l’intérieure de votre tuple (matrice) :

Exemple :

```
tup1 = ('a', 'Lundi', 'b', 1)

# Afficher un élément
print(f"Le premier élément : {tup1[0]}")
print(f"Le dernier élément : {tup1[-1]}")

# Le slicing
print(f"Les deux premiers éléments : {tup1[0:2]}")
print(f"Tout ce qui suit le premier élément : {tup1[1:]}")

# Matrice
mat_tup = ('Lundi', 'Mardi', ('Janvier', 'février'), 'Mercredi')

elem1 = mat_tup[2]
print(f"La liste intérieure : {elem1}")
elem2 = mat_tup[2][0]
```

Résultat :

```
Le premier élément : a
Le dernier élément : 1
Les deux premiers éléments : ('a', 'Lundi')
Tout ce qui suit le premier élément : ('Lundi', 'b', 1)
La liste intérieure : ('Janvier', 'février')
Le premier élément de deuxième tuple : Janvier
```

Exemple :

```
tup = ('Lundi', 'Mardi', [1, 2])

print("Le type de tup :", type(tup))
print(f"Le contenu de tup : {tup}")

# Modification de la liste intérieure
tup[2][0] = 10
print(f"Le contenu de tup après la modification : {tup}")
```

Résultat :

```
Le type de tup : <class 'tuple'>
Le contenu de tup1 : ('Lundi', 'Mardi', [1, 2])
Le contenu de tup après la modification : ('Lundi', 'Mardi', [10, 2])
```

Vérifier si un élément existe

Exemple :

```
tup1 = ('Lundi', 1, 2)

recherche1 = "Lundi" in tup1
recherche2 = "Mardi" in tup1
print(f"Lundi se trouve dans tup1 ? {recherche1}")
print(f"Mardi se trouve dans tup1 ? {recherche2}")
```

Résultat :

```
Lundi se trouve dans tup1 ?  True
Mardi se trouve dans tup1 ?  False
```

Opération sur les tuples

Comme les listes, vous pouvez concaténer et multiplier un tuble.

Quelque fonction utile

Comme les listes, vous pouvez utiliser les fonction len(), max() et min().

Méthodes de l'objets tuple

Méthode	Description
index(x)	Retourne l'indice du premier élément de la valeur x. Si aucun élément ne correspond pas à la valeur x, une erreur s'est produite.
count(x)	Compter le nombre d'occurrences de l'élément x.

Dictionnaires

C’est un Objet qui contient un ensemble de pair d'élément séparées par des virgules et délimiter avec des accolades, les éléments sont des paires clé:valeur séparer par deux points superposés (:) ainsi les clés doivent être uniques et non modifiable.

Avant la version 3.7 de Python, les dictionnaires ne mémorisent pas l'ordre d'insertion des éléments, mais maintenant les dictionnaires devenus ordonnés.

Créer un dictionnaire

Exemple :

```
# Dictionnaire vide
dico1 = {}

# Dictionnaire contenant des éléments
dico2 = {'cle1': 1, 'cle2': 2, 1: 'Lundi'}

# Dictionnaire avec des tuples comme clés
dico3 = {(1, 2): 10, ('jour', 'date', 'mois'): 'Lundi 01 Mars'}

print(f"Le contenu de dico1 : {dico1}")
print(f"Le contenu de dico2 : {dico2}")
print(f"Le contenu de dico3 : {dico3}")
```

Résultat :

```
Le contenu de dico1 : {}
Le contenu de dico2 : {'cle1': 1, 'cle2': 2, 1: 'Lundi'}
Le contenu de dico3 : {(1, 2): 10, ('jour', 'date', 'mois'): 'Lundi 01
Mars'}
```

Afficher un élément

Exemple :

```
dico1 = {'cle1': 10, 'cle2': 50, 1: 'Lundi', 2: 'Mardi'}
print(f"La valeur de la clé 'cle1' : {dico1['cle1']}")

dico1 = {(1, 2): 'position1', (1, 3): 'position2', (2, 1): 'position3'}
print(f"La valeur de la clé (1,2) : {dico1[(1, 2)]}")
```

Résultat :

```
La valeur de la clé 'cle1' : 10
La valeur de la clé (1,2) : position1
```

Ajouter un élément

Pour ajouter des éléments, il faut déterminer une clé avec une valeur.

Exemple :

```
# Dictionnaire vide
dico1 = {}
# Ajouter le premier élément
dico1['cle1'] = 10
print(f"Le contenu de dico1 : {dico1}")
# Ajouter le deuxième élément
dico1['1'] = 50
print(f"Le contenu de dico1 : {dico1}")
# Ajouter le troisième élément
dico1['cle2'] = 'Lundi'
print(f"Le contenu de dico1 : {dico1}")
```

Résultat :

```
Le contenu de dico1 : {'cle1': 10}
Le contenu de dico1 : {'cle1': 10, '1': 50}
Le contenu de dico1 : {'cle1': 10, '1': 50, 'cle2': 'Lundi'}
```

Modifier un élément

Via les clés vous pouvez modifier la valeur d'un élément.

Exemple :

```
dico1 = {'cle1': 10, 'cle2': 50, 1: 'Lundi', 2: 'Mardi'}
print(f"dico1 avant modification : {dico1}")

# Modifier la valeur de cle1
dico1['cle1'] = 2019
print(f"dico1 après modification : {dico1}")
```

Résultat :

```
dico1 avant modification : {'cle1': 10, 'cle2': 50, 1: 'Lundi', 2: 'Mardi'}
dico1 après modification : {'cle1': 2019, 'cle2': 50, 1: 'Lundi', 2: 'Mardi'}
```

Vérifier si une clé existe

Exemple :

```
dico1 = {'cle1': 10, 'cle2': 20, 'cle3': 30}

recherche1 = 'cle1' in dico1
print(f"cle1 se trouve dans dico1 ? {recherche1}")
recherche2 = 'cle4' in dico1
print(f"cle4 se trouve dans dico1 ? {recherche2}")
```

Résultat :

```
cle1 se trouve dans dico1 ?  True
cle4 se trouve dans dico1 ?  False
```

Méthodes de l'objet dictionnaire

Méthode	Description
clear()	Supprimer tous les éléments d'une liste.
get(key,default)	Afficher la valeur de la clé **key**. Si la clé n'existe pas alors soit : +Afficher valeur par défaut fournie **default** +Afficher None si aucune valeur par défaut n'est fournie.
Key()	Afficher tous les clés du dictionnaire.
values()	Afficher tous les valeurs du dictionnaire.
pop(k,default)	Supprimer et afficher l'élément de la clé **key**. Si la clé n'existe pas, alors soit : +Afficher valeur par défaut fournie **default**. +Afficher une erreur si aucune valeur par défaut n'est fournie.
popitem()	Supprime et affiche le dernier pair (clé, valeur). Si le dictionnaire est vide, une erreur s'est produite.
setdefault(k,default)	C'est comme get() mais si la clé n'existe pas et la valeur par défaut fournie, le pair (k, default) est ajouté au dictionnaire.
update(dict2)	Ajouter les éléments de dict2 au dictionnaire s'il y a des clés existe déjà les valeurs sont mises à jour.
copy()	Faire une copie légère de type shallow.

Exemple :

```
dico1 = {'cle1': 10, 'cle2': 20, 'cle3': 30}
dico1.clear()

# Le dictionnaire dico1 après l'utilisation de clear()
print(f"dico1 : {dico1}")
```

Résultat :

```
dico1 : {}
```

Exemple :

```
dico1 = {'cle1': 10, 'cle2': 20, 'cle3': 30}

# Rechercher la valeur de la clé 'cle1'
rch1 = dico1.get('cle1')
print(f"Méthode get sans valeur par défaut : {rch1}")

# Rechercher la valeur de la clé 'cle50'
rch2 = dico1.get('cle50')
print(f"Méthode get sans valeur par défaut : {rch2}")
rch3 = dico1.get('cle50','La clé recherchée n' existe pas')
print(f"Méthode get sans valeur par défaut : {rch3}")
```

Résultat :

```
Méthode get sans valeur par défaut : 10
Méthode get sans valeur par défaut : None
Méthode get sans valeur par défaut : La clé recherchée n' existe pas
```

Exemple :

```
dico1 = {'cle1': 10, 'cle2': 20, 'cle3': 30}

print(f"Les clés de dico1 : {dico1.keys()}")
```

Résultat :

```
Les clés de dico1 : dict_keys(['cle1', 'cle2', 'cle3'])
```

Exemple :

```
dico1 = {'cle1': 10, 'cle2': 20, 'cle3': 30}

print(f"Les valeurs de dico1 : {dico1.values()}")
```

Résultat :

```
Les valeurs de dico1 : dict_values([10, 20, 30])
```

Exemple :

```
dico1 = {'cle1': 10, 'cle2': 20, 'cle3': 30}

# Supprimer l'élément de la clé 'cle1'
val1 = dico1.pop('cle1', 'clé n' existe pas')
print(f"La valeur de la clé 'cle1' qui sera supprimer : {val1}")

# Supprimer l'élément de la clé 'cle50'
val2 = dico1.pop('cle50', 'clé n' existe pas')
print(f"La valeur de 'cle50' qui sera supprimer : {val2}")
```

Résultat :

```
La valeur de la clé 'cle1' qui sera supprimer : 10
La valeur de 'cle50' qui sera supprimer : clé n' existe pas
```

Exemple :

```
dico1 = {'cle1': 10, 'cle2': 20, 'cle3': 30}

# Supprimer le dernier élément du dictionnaire
val1 = dico1.popitem()
print(f"Le dernier élément qui sera supprimer : {val1}")
```

Résultat :

```
Le dernier élément qui sera supprimer : ('cle3', 30)
```

Exemple :

```
dico1 = {'cle1': 10, 'cle2': 20, 'cle3': 30}

print(f"dico1 avant 'setdefault' : {dico1}")

# Supprimer le dernier élément du dictionnaire
rch = dico1.setdefault('cle50', 40)
print(f"La valeur recherchée de la clé 'cle50' : {rch}")
print(f"dico1 après 'setdefault' : {dico1}")
```

Résultat :

```
dico1 avant 'setdefault' : {'cle1': 10, 'cle2': 20, 'cle3': 30}
La valeur recherchée de la clé 'cle50' : 40
dico1 après 'setdefault' : {'cle1': 10, 'cle2': 20, 'cle3': 30, 'cle50': 40}
```

Exemple :

```
dico1 = {'cle1': 10, 'cle2': 20, 'cle3': 30}
dico2 = {'a': 1, 'b': 2}

# Mettre à jour dico1 avec les éléments du dico2
dico1.update(dico2)

print(f"dico1 après la mise à jour : {dico1}")
```

Résultat :

```
dico1 après la mise à jour : {'cle1': 10, 'cle2': 20, 'cle3': 30, 'a': 1,
'b': 2}
```

Copier un dictionnaire

Si votre dictionnaire contient des valeurs non modifiables comme un tuple, alors vous pouvez soit utiliser la méthode copy() ou la fonction dict(). Ces techniques créent des copies légères de type shallow.

Pour faire une copie d'un dictionnaire qui contient une valeur de type liste, vous deviez faire une copie complète (deepcopy). Pour réaliser cette copie vous devez utiliser la fonction deepcopy du module copy.

Mais avant, on va faire une copie légère pour un dictionnaire contenant une valeur de type liste.

Exemple :

```
dico1 = {'cle1': 10, 'cle2': ['a', 'b']}

# Vous pouvez utiliser dico2 = dict(dico1)
dico2 = dico1.copy()

print(f"Le contenu de dico1 : {dico1}")
print(f"Le contenu de dico2 : {dico2}")

# Modification de la liste intérieure du dictionnaire 'dico2'
dico2['cle2'].append('c')

print(f"dico1 après la modification de dico2 : {dico1}")
print(f"dico2 après la modification : {dico2}")
```

Résultat :

```
Le contenu de dico1 : {'cle1': 10, 'cle2': ['a', 'b']}
Le contenu de dico2 : {'cle1': 10, 'cle2': ['a', 'b']}
dico1 après la modification de dico2 : {'cle1': 10, 'cle2': ['a', 'b', 'c']}
dico2 après la modification : {'cle1': 10, 'cle2': ['a', 'b', 'c']}
```

Note : Comme vous remarquez, la modification du dictionnaire ‘dico2’ influence l’autre dictionnaire ‘dico1’.

Copie légère (shallow)

Exemple :

```
dico1 = {1: 'a', 2: 'b'}

dico2 = dico1.copy()
print(f"Le contenu de dico1 : {dico1}")
print(f"Le contenu de dico2 : {dico2}")

# Modification du dictionnaire 'dico2'
dico2[1] = 'Lundi'
print(f"dico1 après la modification de dico2 : {dico1}")
print(f"dico2 après la modification : {dico2}")
```

Résultat :

```
Le contenu de dico1 : {1: 'a', 2: 'b'}
Le contenu de dico2 : {1: 'a', 2: 'b'}
dico1 après la modification de dico2 : {1: 'a', 2: 'b'}
dico2 après la modification : {1: 'Lundi', 2: 'b'}
```

Copie complète (deepcopy)

Exemple :

```
import copy

dico1 = {'cle1': 10, 'cle2': ['a', 'b']}
dico2 = copy.deepcopy(dico1)

print(f"Le contenu de dico1 : {dico1}")
print(f"Le contenu de dico2 : {dico2}")

# Modification de la liste intérieure du dictionnaire 'dico2'
dico2['cle2'].append('c')

print(f"dico1 après la modification de dico2: {dico1}")
print(f"dico2 après la modification : {dico2}")
```

Résultat :

```
Le contenu de dico1 : {'cle1': 10, 'cle2': ['a', 'b']}
Le contenu de dico2 : {'cle1': 10, 'cle2': ['a', 'b']}
dico1 après la modification de dico2: {'cle1': 10, 'cle2': ['a', 'b']}
dico2 après la modification : {'cle1': 10, 'cle2': ['a', 'b', 'c']}
```

Note : avec la fonction deepcopy(), la copie c'est bien passé même si on modifie le dictionnaire dico2 l'autre dictionnaire reste sans modification.

La fonction del

Vous pouvez aussi utiliser la fonction del pour supprimer un élément en indiquant sa clé.

Exemple :

```
dico1 = {'cle1': 'a', 2: 'b', 3: 'c'}

print(f"dico1 avant l'utilisation de la fonction del : {dico1}")

# Supprimer l'élément de la clé 'cle1'
del dico1['cle1']
print(f"dico1 après la suppression de la clé 'cle1' : {dico1}")
```

Résultat :

```
dico1 avant l'utilisation de la fonction del : {'cle1': 'a', 2: 'b', 3: 'c'}
dico1 après la suppression de la clé 'cle1' : {2: 'b', 3: 'c'}
```

Ensembles (set)

C’est un Objet qui contient une collection non ordonnée d'éléments unique non modifiable entouré par des accolades, il est utilisé surtout pour des opérations d’ensembles mathématiques (union, la différence, l’intersection, etc…).

Créer un ensemble(set)

Exemple :

```
mon_set1 = {1, 2, 3, 'a'}

print("Le type de 'mon_set1' :", type(mon_set1))
print(f"Le contenu de 'mon_set1' : {mon_set1}")
```

Résultat :

```
Le type de 'mon_set1' : <class 'set'>
Le contenu de 'mon_set1' : {'a', 1, 2, 3}
```

Vérifier si un élément existe

Exemple :

```
mon_set1 = {1, 2, 3, 'a'}

recherche1 = 'a' in mon_set1
print(f"'a' se trouve dans mon_set1 ? {recherche1}")
recherche2 = 4 in mon_set1
print(f"4 se trouve dans mon_set1 ? {recherche2}")
```

Résultat :

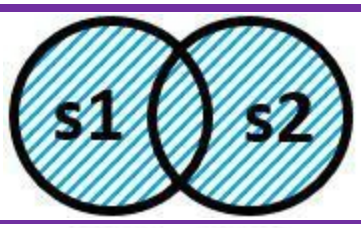
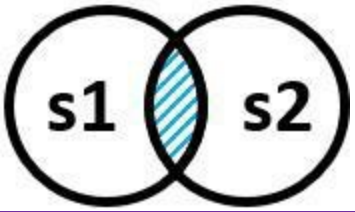
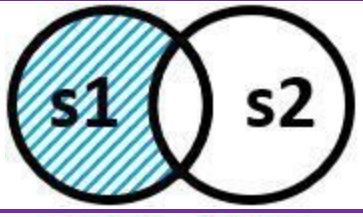
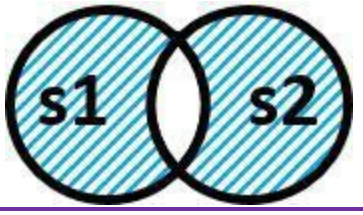
```
'a' se trouve dans mon_set1 ?  True
4 se trouve dans mon_set1 ?  False
```

Méthodes de l'objet set

Méthode	Description
add(x)	Ajouter un seul élément x.
update(x,y,z)	Ajouter plusieurs éléments x,y,z.
discard(x)	Supprime l'élément x, s'il n'existe pas aucune erreur s'affiche.
remove(x)	Supprime l'élément x, s'il n'existe pas une erreur s'affiche.
clear()	Supprimer tous les éléments de l'objet set.
union()	Retourne un objet set contenant les éléments de tous les sets.
Intersection()	Retourne un objet set contenant des éléments partager entre tous les sets.
difference()	Retourne les éléments non contenus dans les autres sets.
symmetric_difference()	Retourne un objet set qui ne contient pas l'intersection entre l'autre sets.
intersection_update()	Supprime les éléments d'un set et les remplacé

	par le résultat de l'intersection.
difference_update()	Supprime les éléments d'un set et les remplacé par le résultat de la différence.
symmetric_difference_update()	Supprime les éléments d'un set et les remplacé par le résultat de la symmetric_difference().
copy()	Créer une copie de l'objet set.

Opération ensembliste

Méthode	Symbole	
union	**s1 | s2**	
Intersection	**s1 & s2**	
difference	**s1 - s3**	
symmetric_difference	**s1 ^ s2**	

Exemple :

```python
s1 = {1, 2, 3}
print(f"Le contenu de 'mon_set1' : {s1}")

# Ajouter un seul élément
s1.add('a')
print(f"'mon_set1' après l'utilisation de add() : {s1}")

# Ajouter plusieurs éléments
s1.update(['b', 'c', 'd'])
print(f"'mon_set1' après l'utilisation de update() : {s1}")
```

Résultat :

```
Le contenu de 's1' : {1, 2, 3}
's1' après l'utilisation de add() : {'a', 1, 2, 3}
's1' après l'utilisation de update() : {1, 2, 'b', 3, 'a', 'c', 'd'}
```

Exemple :

```python
s1 = {1, 2, 3}
print(f"Le contenu de 's1' : {s1}")

# Supprimer un élément existant
s1.discard(1)
print(f"Le contenu de 's1' après la suppression : {s1}")

# Supprimer un élément qui n'existe pas
s1.discard(12)
print(f"Le contenu de 's1' après la suppression : {s1}")
```

Résultat :

```
Le contenu de 's1' : {1, 2, 3}
Le contenu de 's1' après la suppression : {2, 3}
Le contenu de 's1' après la suppression : {2, 3}
```

Exemple :

```
s1 = {1, 2, 3}
# Supprimer un élément qui n'existe pas
s1.remove(12)
print(f"Le contenu de 's1' après la suppression : {s1}")
```

Résultat :

```
Traceback (most recent call last):
  File "D:/Python_cours/code_Python.py", line 3, in <module>
    s1.remove(12)
KeyError: 12
```

Exemple :

```
s1 = {1, 2, 3}
print(f"Le contenu de 's1' avant clear() : {s1}")

# Supprimer tous les éléments
s1.clear()
print(f"Le contenu de 's1' après clear() : {s1}")
```

Résultat :

```
Le contenu de 's1' avant clear() : {1, 2, 3}
Le contenu de 's1' après clear() : set()
```

Exemple :

```
s1 = {1, 2}
s2 = {3, 4}
s3 = {5, 6}

union1 = s1.union(s2)
print(f"La méthode union pour s1 et s2 : {union1}")
union2 = s1.union(s2, s3)
print(f"La méthode union pour s1, s2 et s3 : {union2}")

# Utilisation de l'opérateur |
union3 = s1 | s2
print(f"L'opérateur | pour s1 et s2 : {union3}")
union4 = s1 | s2 | s3
print(f"L'opérateur | pour s1, s2 et s3 : {union4}")
```

Résultat :

```
La méthode union pour s1 et s2 : {1, 2, 3, 4}
La méthode union pour s1, s2 et s3 : {1, 2, 3, 4, 5, 6}
L'opérateur | pour s1 et s2 : {1, 2, 3, 4}
L'opérateur | pour s1, s2 et s3 : {1, 2, 3, 4, 5, 6}
```

Exemple :

```
s1 = {1, 2, 3}
s2 = {1, 3}
s3 = {2, 3, 4}

inter1 = s1.intersection(s2)
print(f"La méthode intersection pour s1 et s2 : {inter1}")
inter2 = s1.intersection(s2, s3)
print(f"La méthode intersection pour s1, s2 et s3 : {inter2}")

# Utilisation de l'opérateur &
inter3 = s1 & s2
print(f"L'opérateur & pour s1 et s2 : {inter3}")
inter4 = s1 & s2 & s3
print(f"L'opérateur & pour s1, s2 et s3 : {inter4}")
```

Résultat :

```
La méthode intersection pour s1 et s2 : {1, 3}
La méthode intersection pour s1, s2 et s3 : {3}
L'opérateur | pour s1 et s2 : {1, 3}
L'opérateur | pour s1, s2 et s3 : {3}
```

Exemple :

```
s1 = {1, 2, 3}
s2 = {1, 3}
s3 = {2, 3, 4}

deffer1 = s1.difference(s2)
print(f"La méthode différence pour s1 et s2 : {deffer1}")
deffer2 = s1.difference(s2, s3)
print(f"La méthode différence pour s1, s2 et s3 : {deffer2}")

# Utilisation de l'opérateur -
deffer3 = s1 - s2
print(f"L'opérateur - pour s1 et s2 : {deffer3}")
deffer4 = s1 - s2 - s3
print(f"L'opérateur - pour s1, s2 et s3 : {deffer4}")
```

Résultat :

```
La méthode différence pour s1 et s2 : {2}
La méthode différence pour s1, s2 et s3 : set()
L'opérateur - pour s1 et s2 : {2}
L'opérateur - pour s1, s2 et s3 : set()
```

Exemple :

```
s1 = {1, 2, 3}
s2 = {1, 3, 6}

sym1 = s1.symmetric_difference(s2)
print(f"symmetric_difference pour s1 et s2 : {sym1}")

# Utilisation de l'opérateur ^
sym2 = s1 ^ s2
print(f"L'opérateur ^ pour s1 et s2 : {sym2}")
```

Résultat :

```
symmetric_difference pour s1 et s2 : {2, 6}
L'opérateur ^ pour s1 et s2 : {2, 6}
```

Exemple :

```
s1 = {1, 2, 3}
s2 = {1, 3, 5}

s1.intersection_update(s2)
print(f"s1 après l'utilisation de intersection_update : {s1}")

# Vous pouvez entrer plus qu' un argument set
# s1.intersection_update(s2, s3)
```

Résultat :

```
s1 après l'utilisation de intersection_update : {1, 3}
```

Exemple :

```
s1 = {1, 2, 3}
s2 = {1, 3}

s1.difference_update(s2)
print(f"s1 après l'utilisation de difference_update : {s1}")

# Vous pouvez entrer plus qu' un argument set
# s1.difference_update(s2, s3)
```

Résultat :

```
s1 après l'utilisation de difference_update : {2}
```

Exemple :

```python
s1 = {1, 2, 3}
s2 = {1, 3, 6}

s1.symmetric_difference_update(s2)
print(f"s1.symmetric_difference_update(s2) : {s1}")

# La méthode symmetric_difference_update()
# accepte un seul argument
```

Résultat :

```
s1.symmetric_difference_update(s2) : {2, 6}
```

Exemple :

```python
s1 = {1, 2, 3}
s2 = s1.copy()

print(f"s1 : {s1}")
print(f"s2 : {s2}")
# Modification de s1
s1.update([50, 'a'])
print(f"s1 après la modification : {s1}")
print(f"s2 après la modification de s1 : {s2}")
```

Résultat :

```
s1 : {1, 2, 3}
s2 : {1, 2, 3}
s1 après la modification : {'a', 1, 2, 3, 50}
s2 après la modification de s1 : {1, 2, 3}
```

Ensembles (frozensets)

Un ensemble (frozenset) c’est comme un ensemble set, sauf qu'il est non modifiable (ensemble figé/gelé) on ne peut pas ajouter, modifier ou supprimer des éléments.

Créer un ensemble frozensets

Pour créer un objet frozensets il suffit de convertir une liste, tuple ou une set par la fonction frozensets().

Exemple :

```
s1 = frozenset([1, 2, 3])

print("Le type de s1 :",type(s1))
print(f"Le contenu de l'ensemble gelé : {s1}")
```

Résultat :

```
Le type de s1 : <class 'frozenset'>
Le contenu de l'ensemble gelé : frozenset({1, 2, 3})
```

Exemple :

```
s2 = frozenset({'Lundi', 'Mardi', 30})

print("Le type de s2 :",type(s2))
print(f"Le contenu de l'ensemble gelé : {s2}")
```

Résultat :

```
Le type de s2 : <class 'frozenset'>
Le contenu de l'ensemble gelé : frozenset({'Lundi', 30, 'Mardi'})
```

Méthodes de l’objet frozensets

Comme les ensembles, frozenset accepte les méthodes suivant : union(), intersection(), difference(), symmetric_difference(), copy().

Exemple :

```
s1 = frozenset([1, 2, 3])
s2 = frozenset([1, 2, 4])

union1 = s1.union(s2)
print(f"La méthode union pour s1 et s2 : {union1}")

# Utilisation de l'opérateur |
union2 = s1 | s2
print(f"Utilisation de l'opérateur | pour s1 et s2 : {union2}")
```

Résultat :

```
La méthode union pour s1 et s2 : frozenset({1, 2, 3, 4})
Utilisation de l'opérateur | pour s1 et s2 : frozenset({1, 2, 3, 4})
```

Comparaison entre les Collections

Collection	Ordonnée	Indexée	Modifiable	Doublons
liste	Oui	Oui	Oui	Oui
Tuple	Oui	Oui	Non	Oui
dictionnaire	Oui	Oui (Clé)	Oui	Non
Set	Non	Non	Oui	Non
frozenset	Non	Non	Non	Non

La fonction input()

Nous avons vue précédemment la fonction print() qui sert de faire une sortie d'information sous forme d'un message texte, mais pour faire l'inverse c’est-à-dire entrer des informations, on va utiliser la fonction input().

L’exécution de la fonction input() suspend l'exécution du programme et attend l'utilisateur pour qu'il entre une valeur et valider par la touche entrée, après la fonction input() stocke la valeur entrer par l'utilisateur dans une variable.

Le type de valeur retourner par la fonction input() est toujours de type chaine de caractères (string), alors il faut faire attention si vous voulez récupérez des valeurs numériques dans, ce cas il faut poncer de faire la conversion.

Exemple :

```
var1 = input("Quel est votre nom ? ")
print(f"La valeur entrée : {var1}")
```

Résultat :

```
Quel est votre nom ?
```

Comme vous voyez, le message est affiché et attend que nous entrons une valeur. Donc on va écrire comme exemple le mot Python et valider par la touche clavier Entrée.

```
Quel est votre nom ? Python
```

Après la validation, notre programme continu l’exécution et affiche la valeur entrer par l’utilisateur via la fonction print().

```
Quel est votre nom ? Python
La valeur entrée : Python
```

Exemple :

```
var1 = input("Entrer le premier nombre : ")
var2 = input("Entrer le deuxième nombre : ")
somme = var1 + var2
print(f"La somme de {var1} et {var2} égale : {somme}")
```

Résultat :

```
Entrer le premier nombre : 5
Entrer le deuxième nombre : 3
La somme de 5 et 3 égale : 53
```

Vous remarquez bien sûr que 5 + 3 n'égale pas 53, tous simplement par ce que la fonction input() retourne des valeurs de type chaine de caractères donc la solution c'est convertie les deux variables.

Exemple :

```
var1 = input("Entrer le premier nombre : ")
var2 = input("Entrer le deuxième nombre : ")
somme=int(var1)+int(var2)
print(f"La somme de {var1} et {var2} égale : {somme}")
```

Résultat :

```
Entrer le premier nombre : 5
Entrer le deuxième nombre : 3
La somme de 5 et 3 égale : 8
```

Contrôle de flux

L'instruction if

L’instruction if (si en français) permet de vérifier la validité d'une condition, dans le cas où la condition est vraie alors le bloc d’instruction qui suit sera exécuté sinon il sera sauté.

L’instruction if est composé par le mot réservé if suivi par une condition puis par deux points superposés (:), et dans la ligne suivante qui doit obligatoirement commencer par une tabulation il y aura le bloc d'instructions à exécuter si la condition est vraie.

Vous pouvez ajouter une autre partie contenant un bloc d’instruction dans le cas où l’instruction if est fausse ; cette partie est composé par le mot réservé else puis par deux points superposés (:).

Vous pouvez ajouter aussi une autre condition qui va-t-être vérifier si la première condition est fausse ; cette partie est composé par le mot réservé elif suivi par une condition puis par deux points superposés (:).

Note : voire les parties précédentes Opérateurs de comparaison et Opérateurs logiques.

Syntaxe 1 :

```
if condition:
    bloc1 d'instruction
    bloc2 d'instruction
```

Syntaxe 2 :

```
if condition:
    bloc1 d'instruction
    bloc2 d'instruction
else:
    bloc3 d'instruction
```

Syntaxe 3 :

```
if condition1:
    bloc1 d'instruction
elif condition2:
    bloc2 d'instruction
else:
    bloc3 d'instruction
```

Syntaxe 4 : Instructions imbriquées

```
if condition1:
    if condition2:
        bloc1 d'instruction
    else:
        bloc2 d'instruction
elif condition3:
    if condition4:
        bloc3 d'instruction
    elif condition5:
        bloc4 d'instruction
    else:
        bloc5 d'instruction
else:
    bloc6 d'instruction
```

Exemple :

```
x = 10

if x > 5:
    print("x est strictement supérieur à 5")
```

Résultat :

```
x est strictement supérieur à 5
```

Exemple :

```
x = 10

if x == 10:
    print("x égal à 10")
```

Résultat :

```
x égal à 10
```

Exemple :

```
x = 10

if x < 5:
    print("x est strictement inférieur à 5")
elif x == 5:
    print("x égal à 5")
else:
    print("x est strictement supérieur à 5")
```

Résultat :

```
x est strictement supérieur à 5
```

Exemple :

```
# ---------------Partie 1---------------
cls = input("Entrer le nom de votre classe (A ou B) : ")
gr = input("Entrer le numéro de groupe (1 ou 2) : ")
# ---------------Partie 2---------------
classe = cls.upper()
groupe = int(gr)
# ---------------Partie 3---------------
if classe == "A":
    if groupe == 1:
        print("Votre numéro de salle est 1")
    else:
        print("Groupe introuvable !")
elif classe == "B":
    if groupe == 1:
        print("Votre numéro de salle est 2")
    elif groupe == 2:
        print("Votre numéro de salle est 3")
    else:
        print("Groupe introuvable !")
else:
```

Partie 1 : l'utilisateur doit entrer le nom d'une classe et le numéro de groupe.

Partie 2 : vue que Python est sensible à la casse on a utilisé la méthode upper() pour le nom de classe, aussi on a converti le groupe vers le type numérique pour faire la comparaison.

Partie 3 : on commence la vérification des classes si « A » ou « B » si non il n'y a pas de classe trouvé après si l'un des premières conditions et vrais l'étape suivante c'est la vérification des groupes.

Alors voyons les résultats pour des diffèrent cas :

Cas 1 : classe et groupe bien trouvé

```
Entrer le nom de votre classe (A ou B) : a
Entrer le numéro de groupe (1 ou 2) : 1
Votre numéro de salle est 1
```

```
Entrer le nom de votre classe (A ou B) : B
Entrer le numéro de groupe (1 ou 2) : 2
Votre numéro de salle est 3
```

Cas 2 : classe trouvé mais pas pour le groupe

```
Entrer le nom de votre classe (A ou B) : a
Entrer le numéro de groupe (1 ou 2) : 2
Groupe introuvable !
```

Cas 3 : classe non trouvé

```
Entrer le nom de votre classe (A ou B) : C
Entrer le numéro de groupe (1 ou 2) : 1
Classe introuvable !
```

L’instruction if avec and et or

Vous pouvez utiliser les Opérateurs logiques dans l’instruction if.

Exemple :

```
nbr = 20

if nbr > 10 and nbr < 100:
    print("Les deux conditions sont vrais")
```

Résultat :

```
Les deux conditions sont vrais
```

Exemple :

```
nbr = 20

if nbr > 10 or nbr == 5:
    print("L'un des conditions est vrais")
```

Résultat :

```
L'un des conditions est vrais
```

La forme courte de l’instruction if

Vous pouvez sur la même ligne mètre l'instruction if et le bloc d'instruction à exécuter.

Exemple :

```
a = 5

if a > 0: print("Le nombre est positif")
```

Résultat :

```
Le nombre est positif
```

Exemple :

```
a = 5

print("a négatif") if a < 0 else print("a positif")
```

Résultat :

```
a positif
```

Exemple :

```
a = 0

print("a négatif") if a < 0 else print("a positif") if a > 0 else print("a égale 0")
```

Résultat :

a égale 0

Les boucles

Une boucle (en anglais loop) ou une instruction répétitive vous permet de répéter des instructions selon vos besoins.

Il y a deux instructions répétitive en Python ; while et for.

Note : voire la partie précédente Opérateurs d’affectation

while

L'instruction while (« tant que » en français) est utilisée pour exécuter des instructions de manière répétée tant qu'une expression est vraie.

Syntaxe :

```
while condition:
    bloc1 d'instructions
    bloc2 d'instructions
```

Exemple :

```
nbr = 1
while nbr <= 4:
    print(f"Le nombre égale : {nbr}")
    nbr += 1
```

Résultat :

```
Le nombre égale : 1
Le nombre égale : 2
Le nombre égale : 3
Le nombre égale : 4
```

Exemple :

```python
# Table de multiplication par 5
nbr = 5
a = 1

while a <= 10:
    print(f"{nbr} * {a} = {nbr * a}")
    a += 1
```

Résultat :

```
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

Exemple :

```
lst1 = ['Lundi', 'Mardi', 'Mercredi', 'Jeudi', 'Vendredi']

# calcule la longueur de la liste
long = len(lst1)
print(f"La longueur de la liste = {long}")

# Parcourir les éléments de la liste
j = 0
while j < long:
    print(f"L'indice = {j} et l'élément = {lst1[j]}")
    j += 1
```

Résultat :

```
La longueur de la liste = 5
L'indice = 0 et l'élément = Lundi
L'indice = 1 et l'élément = Mardi
L'indice = 2 et l'élément = Mercredi
L'indice = 3 et l'élément = Jeudi
L'indice = 4 et l'élément = Vendredi
```

for

La boucle for permet de parcourir les éléments d'une séquence, comme une liste par exemple.

Syntaxe :

```
for x in séquence:
    bloc d'instruction
    bloc d'instruction
```

Exemple :

```
ls1 = [1, 2, 3]

for x in ls1:
    print(x)
```

Résultat :

```
1
2
3
```

Exemple :

```python
lst1 = ['Lundi', 'Mardi', 'Mercredi', 'Jeudi', 'Vendredi']

# Parcourir les éléments d'une liste
for elem in lst1:
    print(elem)
```

Résultat :

```
Lundi
Mardi
Mercredi
Jeudi
Vendredi
```

Exemple :

```python
lst1 = ['Lundi', 'Mardi', 'Mercredi', 'jeudi', 'vendredi']

# calcule la longueur de la liste
long = len(lst1)
print(f"La longueur de la liste = {long}")

# Parcourir les éléments de la liste
for indice in range(len(lst1)):
    print(f"L'indice = {indice} et l'élément = {lst1[indice]}")
```

Résultat :

```
La longueur de la liste = 5
L'indice = 0 et l'élément = Lundi
L'indice = 1 et l'élément = Mardi
L'indice = 2 et l'élément = Mercredi
L'indice = 3 et l'élément = jeudi
L'indice = 4 et l'élément = vendredi
```

Exemple :

```
dict1 = {'cle1':'Lundi', 'cle2':'Mardi', 'cle3':'Mercredi'}

# Parcourir les éléments d'un dictionnaire
for cle in dict1:
    print(f"{cle} : {dict1[cle]}")
```

Résultat :

```
cle1 : Lundi
cle2 : Mardi
cle3 : Mercredi
```

L’instruction else

Vous pouvez ajouter l'instruction else à la boucle while ou for, qui va s'exécuter quand les boucles seront terminées ou la condition de while devient fausse.

Exemple :

```
a = 3

while a >= 1:
    print(a)
    a -= 1
else:
    print("Compte à rebours terminé")
```

Résultat :

```
3
2
1
Compte à rebours terminé
```

Exemple :

```
ls1 = [1, 2, 3]

for x in ls1:
    print(x)
else:
    print("Tous les éléments sont affichés")
```

Résultat :

```
1
2
3
Tous les éléments sont affichés
```

Les boucles imbriquées

Vous pouvez faire une loupe à l'intérieur d'une autre loupe comme l'exemple suivant :

Exemple :

```
ls1 = [0, 1]
ls2 = [0, 1]

for x in ls1:
    for y in ls2:
        print(f"x= {x}, y={y}")
```

Résultat :

```
x= 0, y=0
x= 0, y=1
x= 1, y=0
x= 1, y=1
```

range()

La fonction range() génère une liste des nombres de valeurs croissantes. Il y a 3 façon décrire la fonction range :

- range(n) : une liste contiendra n nombre qui commence par le numuro 0
- range(x, n) : une liste qui commence par le numéro x et terminer par n-1
- range(x, n, step) : une liste qui commence par le numéro x, terminer par n-1 et le STEP «pas» à sauter qui est par défaut c'est 1 mais vous pouvez entrer par exemple 2 si vous voulez que votre liste s'incrémente par 2 nombre (1,3,5,7,...)

Exemple :

```
# range() avec un seul paramètre
for x in range(3):
    print(x)
```

Résultat :

```
0
1
2
```

Exemple :

```
# range avec deux paramètres
for x in range(1, 5):
    print(x)
```

Résultat :

```
1
2
3
4
```

Exemple :

```
# range avec trois paramètres
for x in range(1, 10, 3):
    print(x)
```

Résultat :

```
1
4
7
```

Les instructions break et continue

- break : arrêter une boucle avant sa fin
- continue : retourne directement au début de la boucle sans exécuter le reste des instructions

Exemple :

```
for i in range(6):
    if i == 3:
        break
    print(i)
```

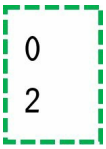
Résultat :

```
0
1
2
```

Exemple :

```
for i in range(3):
    if i == 1:
        continue
    print(i)
```

Résultat :


0
2

Exemple :

```
# "while 1:" c'est une boucle infinie
# par ce que 1 est toujours vrai
# "while True:" c'est aussi une boucle infinie

while 1:
    mot = input("Tapez 'fin' pour quitter : ")
    if mot == "fin":
        print("Mot correct")
        break
```

Résultat :

```
Tapez 'fin' pour quitter : Fin
Tapez 'fin' pour quitter : 100
Tapez 'fin' pour quitter : fin
Mot correct
```

Les fonctions

C’est un bloc d'instruction exécuté seulement lorsqu'il est appelé. Nous avons déjà vue des fonctions comme print() et input(), mais nous allons créer nos propres fonctions.

Syntaxe :

```
def nom_fonction(liste des paramètres):
    bloc d’instruction
    return resultat
```

def	C’est le mot clé qui permet de définir une fonction, La ligne se termine par un deux points superposés (:)
nom_fonction	Le nom de la fonction suit les mêmes règles de la création d'une variable.
liste des paramètres	Un paramètre c'est l'information fourni à la fonction, séparé par des virgules et entouré par des parenthèses, une fonction peut être créer sans paramètre mais il est obligatoire d’avoir des parenthèse vide.
Return	C’est un mot clé qui permet d'arrêter l'exécution de la fonction et retourné une valeur, non obligatoire dans le cas d'une fonction qui ne retourne pas des résultats.

Appeler une fonction

Pour exécuter ou appeler une fonction il suffit d'écrire son nom. Il est aussi possible qu’une fonction appelle une autre fonction.

Commentaire interne d'une fonction

Juste la ligne qui suit la définition de la fonction, vous pouvez ajouter une chaine de caractères entourée par des guillemets ("), cette chaine est traitée comme un commentaire interne de la fonction stocké dans une variable spéciale nommée __doc__.

Fonction sans paramètres

Exemple :

```
def msg_fonction():
    print("Bonjour tout le monde !")


msg_fonction()  # Appeler la fonction
```

Résultat :

```
Bonjour tout le monde !
```

Exemple :

```
def msg():
    print("Bonjour tout le monde !")


def repeter_msg():
    "Appeler la fonction msg() 2 fois"
    msg()
    msg()


repeter_msg()  # Appeler la fonction
```

Résultat :

```
Bonjour tout le monde !
Bonjour tout le monde !
```

Exemple :

```
def msg_fonction():
    msg = "Bonjour tout le monde !"
    return msg  # Envoie de la valeur


# Récupérer la valeur envoyer par la fonction
resultat = msg_fonction()
print(resultat)  # Afficher la valeur récupérée
```

Résultat :

```
Bonjour tout le monde !
```

Exemple :

```
def msg_fonction():
    "Fonction sans paramètre qui affiche un message"
    print("Bonjour tout le monde !")


# Afficher le commentaire interne de la fonction
print(msg_fonction.__doc__)
# Appeler la fonction
msg_fonction()
```

Résultat :

```
Fonction sans paramètre qui affiche un message
Bonjour tout le monde !
```

Fonction avec paramètres

Exemple :

```
def msg_fonction(msg):
    print(msg)


# Appeler la fonction avec une valeur de paramètre
msg_fonction("Bonjour tout le monde !")
```

Résultat :

```
Bonjour tout le monde !
```

Exemple :

```
def somme_fonction(nbr1, nbr2):
    "Envoyer la somme de nbr1 et nbr2"
    return nbr1 + nbr2


# Appeler la fonction avec deux argument : 5 et 8
# Récupérer le résultat envoyé par la fonction
somme = somme_fonction(5, 9)
print(f"La somme de 5 et 9 égale : {somme}")
```

Résultat :

```
La somme de 5 et 9 égale : 14
```

Exemple :

```
def test(nbr):
    if nbr > 0:
        msg = "Nombre positif"
    elif nbr < 0:
        msg = "Nombre négatif"
    else:
        msg = "Nombre égale zéro"
    return msg


print("Le premier test :", test(0))
print("Le deuxième test :", test(5))
print("Le troisième test :", test(-8))
```

Résultat :

```
Le premier test : Nombre égale zéro
Le deuxième test : Nombre positif
Le troisième test : Nombre négatif
```

Exemple :

```
def total(param):
    somme = 0
    for nbr in param:
        somme += nbr
    return somme


# Création d'un tuple qui contient des valeurs
tuple1 = [1, 2, 3, 4, 5]
# Fonction appelée avec un tuple comme argument
resultat = total(tuple1)
print(f"Résultat d'envoi d'un tuple : {resultat}")
```

Résultat :

```
Résultat d'envoi d'un tuple : 15
```

Exemple :

```
def affichage(param):
    for cle in param:
        print(f"{cle} : {param[cle]}")


# Création d'un dictionnaire
dict = {'cle1': 1, 'cle2': 'Python'}
affichage(dict)
```

Résultat :

```
cle1 : 1
cle2 : Python
```

Fonctions de rappel (callbacks)

Vous pouvez mettre comme paramètre le nom d'une fonction, voyons un exemple pour bien comprendre.

Exemple :

```
def somme(x, y):
    return x + y

def produit(x, y):
    return x * y

def calcul(x, y, f):
    return f(x, y)

# Appelle direct de la fonction somme
print("La fonction somme(5, 8) :", somme(5, 8))
# appelle direct de la fonction produit
print("La fonction produit(5, 8) :", produit(5, 8))
# La fonction calcul avec nom d'une fonction comme argument
print("La fonction calcul(5, 8, somme) :", calcul(5, 8, somme))
print("La fonction calcul(5, 8, produit) :", calcul(5, 8, produit))
```

Résultat :

```
La fonction somme(5, 8) : 13
La fonction produit(5, 8) : 40
La fonction calcul(5, 8, somme) : 13
La fonction calcul(5, 8, produit) : 40
```

Paramètre formel

Vous pouvez fournir autant de valeurs à votre fonction sous la forme (*parametre) avec une seule étoile ou (**parametre) avec deux étoile qui reçoit des valeurs nommées sous forme des pair (nom=valeur).

Exemple :

```
def total(*param):
    return param[0] + param[1] + param[2]


resultat1 = total(1, 2, 3)
print(f"Résultat d'envoi de 3 paramètres : {resultat1}")
resultat2 = total(8, 30, 50)
print(f"Résultat d'envoi de 3 paramètres : {resultat2}")
```

Résultat :

```
Résultat d'envoi de 3 paramètres : 6
Résultat d'envoi de 3 paramètres : 88
```

Exemple :

```python
def parcourir(*param):
    for elem in param:
        print(elem)


parcourir(1, 2, 3)
```

Résultat :

```
1
2
3
```

Exemple :

```python
def affichage(**param):
    print(f"La valeur de cle1 : {param['cle1']}")
    print(f"La valeur de cle2 : {param['cle2']}")


# Appeler la fonction avec des paramètres
affichage(cle1=10, cle2="Python")
```

Résultat :

```
La valeur de cle1 : 10
La valeur de cle2 : Python
```

Exemple :

```
def parcourir(**param):
    for cle in param:
        print(f"{cle} : {param[cle]}")


parcourir(para1=1, para2='Lundi')
```

Résultat :

```
para1 : 1
para2 : Lundi
```

return multiple

Les fonctions peuvent retourner plus qu'une valeur, regrouper dans une liste, tuple ou dictionnaire.

Exemple :

```
def double(x, y):
    return x * 2, y * 2


# Récupérer les valeurs envoyées
val1, val2 = double(5, 8)
print(f"val1 = {val1}, val2 = {val2}")
```

Résultat :

```
val1 = 10, val2 = 16
```

Exemple :

```
def double(x, y):
    return x * 2, y * 2


# Récupérer le tuple envoyé
tup1 = double(5, 8)
print("Type des valeurs envoyées :", type(tup1))
print(f"Le contenu du tuple : {tup1}")
```

Résultat :

```
Type des valeurs envoyées : <class 'tuple'>
Le contenu du tuple : (10, 16)
```

Exemple :

```python
def double(x, y):
    return  [x * 2, y * 2]


# Récupérer la liste envoyée
lst1 = double(5, 8)
print("Type des valeurs envoyées :", type(lst1))
print(f"Le contenu du liste : {lst1}")
```

Résultat :

```
Type des valeurs envoyées : <class 'list'>
Le contenu du liste : [10, 16]
```

Exemple :

```
def double(x, y):
    return {'val1': x * 2, 'val2': y * 2}


# Récupérer le dictionnaire envoyé
dict1 = double(5, 8)
print("Type des valeurs envoyées :", type(dict1))
print(f"Le contenu du dictionnaire : {dict1}")
```

Résultat :

```
Type des valeurs envoyées : <class 'dict'>
Le contenu du dictionnaire : {'val1': 10, 'val2': 16}
```

Ordre des arguments

Quand vous appelez une fonction, les arguments fournis doit correspond exactement au même ordre avec les paramètres.

Mais vous pouvez fournir les arguments avec n'importe quel ordre à condition de spécifier chaque paramètre avec l'argument qui lui correspond.

Exemple :

```
def soustraction(nbr1, nbr2):
    return nbr1 - nbr2


print("nbr1 reçoit 8 et nbr2 reçoit 5 :", soustraction(8, 5))
print("nbr1 reçoit 5 et nbr2 reçoit 8 :", soustraction(5, 8))
# On précise chaque paramètre avec leur argument
resultat = soustraction(nbr2=5, nbr1=8)
print(f"Chaque paramètre avec leur argument : {resultat}")
```

Résultat :

```
nbr1 reçoit 8 et nbr2 reçoit 5 : 3
nbr1 reçoit 5 et nbr2 reçoit 8 : -3
Chaque paramètre avec leur argument : 3
```

Valeur par défaut des paramètres

Quand vous créez une fonction, il est possible de définir des arguments par défaut pour chaque paramètre.

Si vous appelez une fonction avec des paramètres manquants, vous obtiendrez une erreur, mais si vous déterminez les valeurs par défaut, les paramètre non entrer garde la valeur par défaut déjà affectée.

Exemple :

```
def message(nom):
    msg = "Bonjour " + nom
    print(msg)


message()
```

Résultat :

```
Traceback (most recent call last):
  File "D:/Python_cours/code_Python.py", line 6, in <module>
    message()
TypeError: message() missing 1 required positional argument: 'nom'
```

Exemple :

```
def message(nom="tout le monde !"):
    msg = "Bonjour " + nom
    print(msg)


# Fonction appelée sans argument
message()
# Fonction appelée avec argument
message("Python")
```

Résultat :

```
Bonjour tout le monde !
Bonjour Python
```

Variable locale et globale

Lorsque vous créez une fonction tous les variables situées à l'intérieur de cette fonction appelées variables locales, ces variables est accessible seulement par la fonction et quand la fonction sera terminée tous les variables locales seront détruites.

Les variables externes de la fonction se sont des variables globales accessible par tout notre programme, vous pouvez à tout moment d'avoir des variables globales à l'intérieur de votre fonction via l'instruction "global".

Exemple :

```
def nombre():
    "Variable locale x"
    x = 2
    print(f"La valeur locale de x : {x}")


# Créer une variable x avec valeur 50
x = 50
print(f"x avant l'appelle de la fonction : {x}")
# Appeler la fonction
nombre()
print(f"x après l'appelle de la fonction : {x}")
```

Résultat :

```
x avant l'appelle de la fonction : 50
La valeur locale de x : 2
x après l'appelle de la fonction : 50
```

Exemple :

```
def nombre():
    "Définir la variable globale x"
    global x
    x = 2
    print(f"La valeur globale de x : {x}")


# Créer une variable x avec valeur 50
x = 50
print(f"x avant l'appelle de la fonction : {x}")
# Appeler la fonction
nombre()
print(f"x après l'appelle de la fonction : {x}")
```

Résultat :

```
x avant l'appelle de la fonction : 50
La valeur globale de x : 2
x après l'appelle de la fonction : 2
```

Fonction lambda

C’est une fonction anonyme extrêmement courte qui contient une seule instruction, le mot-clé "lambda" est utilisé pour la création de cette fonction.

Syntaxe :

```
lambda paramétre1, paramétre2, ⋯: instruction de retour
```

Exemple :

```
f = lambda x, y: x * y

print("Appeler la fonction lambda f(5, 7) :", f(5, 7))
print("Appeler la fonction lambda f(-2, 5) :", f(-2, 5))
```

Résultat :

```
Appeler la fonction lambda f(5, 7) : 35
Appeler la fonction lambda f(-2, 5) : -10
```

Les modules

Un module c'est un fichier avec l'extension .py contenant des instructions comme les fonctions. Quand vous voulez utilisé l’un des fonctions enregistrer dans un module, vous pouvez l’importé vers votre fichier de code, on vira par la suite comment le faire. Mais d’abord nous devons savoir comment et où créer un module.

Localisation des modules

Par défaut Python recherche les modules selon l'ordre suivant :

1. Le répertoire en cours où il est enregistré votre projet
2. Si le module n'existe pas dans le répertoire en cours, Python recherche dans chaque répertoire situé dans la variable shell PYTHONPATH
3. La dernière étape, c'est la recherche dans le chemin par défaut qui est le dossier d'installation de Python

Lieu des modules créer

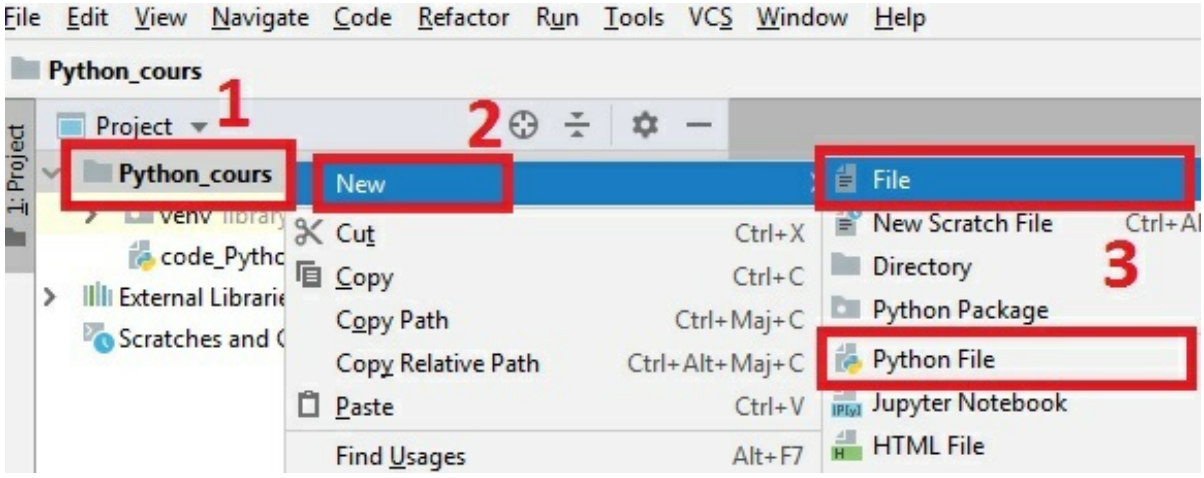
Il est préférable d'enregistrer vos modules dans le répertoire courant qui contient votre projet.

Créer un module

Dans le répertoire de votre projet créer un fichier avec l'extension .py et ajouter les instructions souhaitées et enregistrer le fichier.

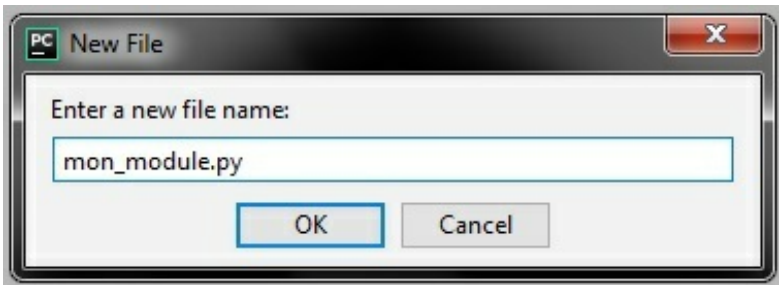
Si vous utilisez un éditeur simple il suffit d'ouvrir le dossier contenant le fichier script et ajouter à coter de lui un nouveau fichier avec l'extension .py

Mais si vous utilisez PyCharme comme moi, suivi les étapes suivant :

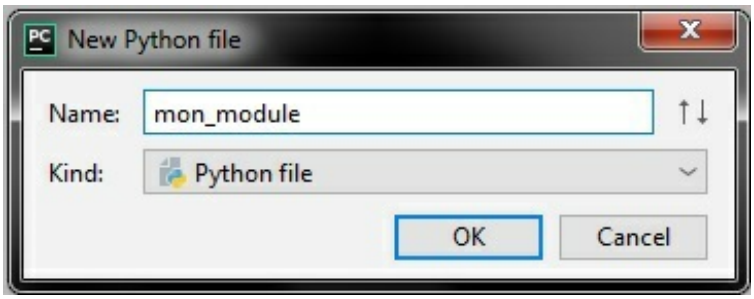


1. Clic droit sur le répertoire de votre projet
2. Sélectionner New
3. Sélectionner soit File ou Python File, la différence c'est que la première tu dois écrire manuellement l'extension .py par contre la deuxième c'est ajouter automatiquement il suffit d'écrire seulement le nom du module

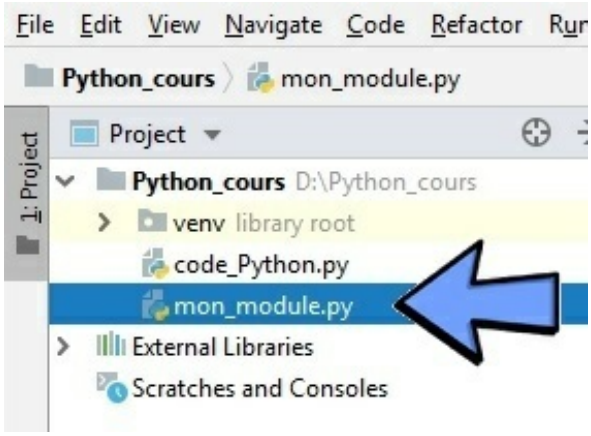
Si vous sélectionnez File, la boite de dialogue suivante s'affiche, remarqué que j'ai ajouté l'extension .py



Si vous sélectionnez le deuxième choix Python File il suffit d'entrer le nom du module sans extension comme montré ci-dessous :



Après la validation remarquez l'ajout d'un fichier « mon_module.py » dans notre répertoire « Python_cours ».



Utiliser un module

Dans le module créé mon_module.py on va ajouter deux fonction :

Exemple :

```
def somme(x, y):
    "La somme de deux nombres"
    return x + y


def produit(x, y):
    "La produit de deux nombres"
    return x * y
```

Après la création des deux fonctions, on va ouvrir le fichier script code_Python.py et ajouter au début le mot réservé suivant :

- import « nom de module »

```
import mon_module
```

Après cette commande, tous les fonctions qui existe dans le module « mon_module » seront disponible, il vous suffit d'écrire le nom du module suivi d'un point puis le nom de la fonction.

Exemple :

```
import mon_module

sm1 = mon_module.somme(2, 5)
print(f"Le résultat de la fonction somme : {sm1}")
```

Résultat :

```
Le résultat de la fonction somme : 7
```

Vous pouvez donnée un nom différant au module importer via le mot réservé « as » suivi par le nom que vous voulez l'utiliser dans le fichier script, noter bien que cette nomination n'affecte pas le module c'est juste une alias.

Exemple :

```
import mon_module as mod

sm1 = mod.produit(5, 6)
print(f"Le résultat de la fonction produit : {sm1}")
```

Résultat :

```
Le résultat de la fonction produit : 30
```

Il existe une autre format d'importation qui permet de déterminer les fonctions que vous voulez l'importer et non tout le module entier.

Exemple :

```
# Importer la fonction somme
from mon_module import somme
```

Exemple :

```
# Importer tous les fonctions Avec le « joker » *
from mon_module import *
```

Exemple :

```
from mon_module import somme

sm1 = somme(5, 2)
print(f"Le résultat de la fonction somme : {sm1}")
```

Résultat :

```
Le résultat de la fonction somme : 7
```

Exemple :

```
from mon_module import *

sm1 = produit(5, 2)
print(f"Le résultat de la fonction produit : {sm1}")
```

Résultat :

```
Le résultat de la fonction produit : 10
```

Commentaire interne d'un module

Lorsqu'on écrit un module il est important d'ajouter des commentaires pour le module et pour chaque fonction pour donner une explication soit pour le module lui-même ou pour chaque fonction, on a déjà vue comment commenté une fonction et pour un module c'est le même ; c'est ajouter une chaine de caractères entourée par des guillemets (") au début du module avant tous les autres instructions.

Alors pour voir les commentaire c'est simple, il faut utiliser la fonction help().

D'abord on ajoute un commentaire à notre module :

Exemple :

```
"Module pour la somme et le produit de deux variable"

def somme(x, y):
    "La somme de deux nombres"
    return x + y


def produit(x, y):
    "La produit de deux nombres"
    return x * y
```

Dans le fichier script « code_Python.py » on import le module « mon_module.py ».

Exemple :

```
import mon_module

print(help(mon_module))
```

Résultat :

```
Help on module mon_module:

NAME
    mon_module - Module pour la somme et le produit de deux variable

FUNCTIONS
    produit(x, y)
        La produit de deux nombres

    somme(x, y)
        La somme de deux nombres

FILE
    d:\python_cours\mon_module.py
```

La fonction dir()

Cette fonction affiche une liste de tous les fonctions d'un module donné.

Exemple :

```
import mon_module

print(dir(mon_module))
```

Résultat :

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', 'produit', 'somme']
```

Les packages

C’est un dossier contenant un ensemble des modules. L’objectif c’est l’organisation de votre projet, on peut avoir aussi des packages qui contient d’autre packages.

Créer packages

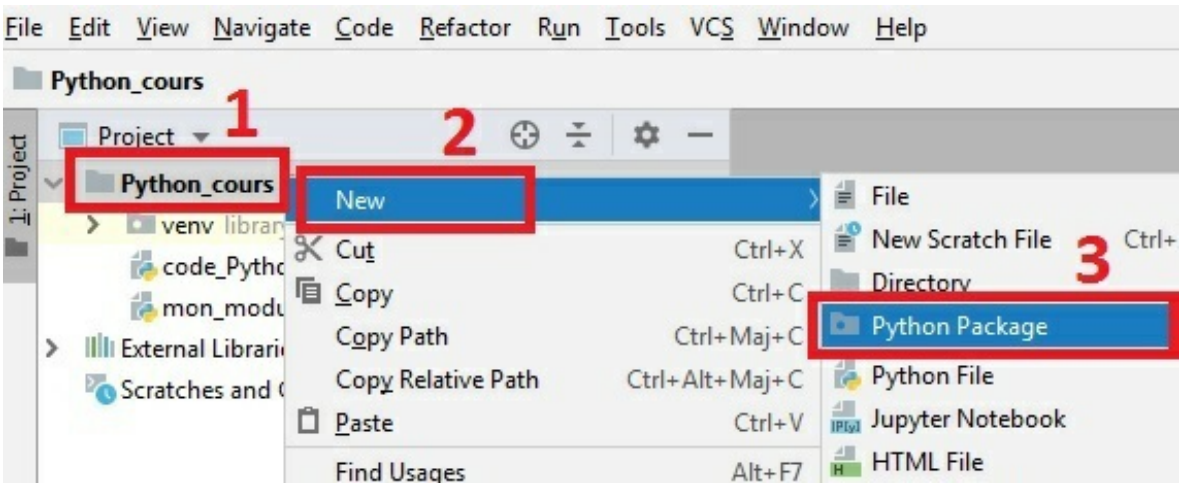
Dans le répertoire de votre projet, créer un dossier qui porte le nom de votre package, ce nom ne doit pas contient des caractères spéciaux ou des espaces (une exception c'est fait pour le caractère souligné _).

Après la création, ouvrir le dossier et ajouter un fichier qui porte le nom « __init__.py » pour que Python reconnaisse ce dossier comme un package, pour l'instant ce fichier peut être vide.

Dans ce répertoire on peut mettre :

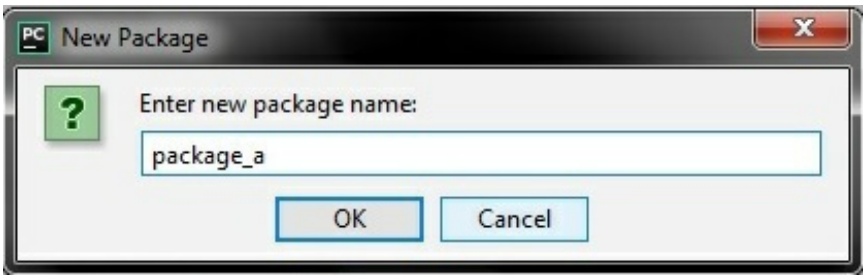
- Des modules
- Des sous-package (chaque dossier doit contenir un fichier « __init__.py » pour que Python reconnaisse ce dossier comme un package)

Si vous utilisez PyCharme comme moi, suivi les étapes suivantes :

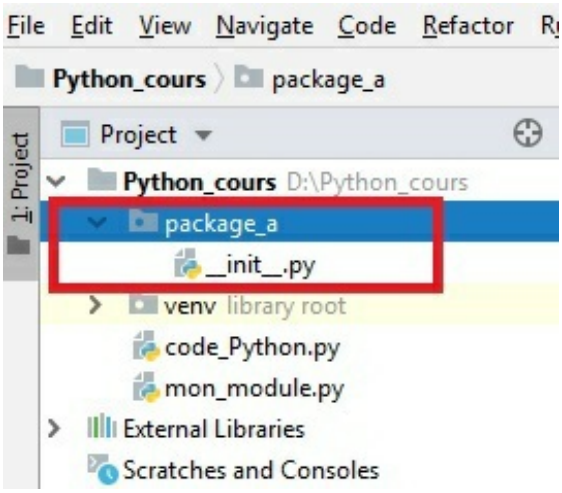


1. Clic droit sur le répertoire de votre projet
2. Sélectionner New
3. Sélectionner Python Package

Une boite de dialogue s'affiche qui demande l'entrer d'un nom du package comme il est montré dans l'image au-dessous :



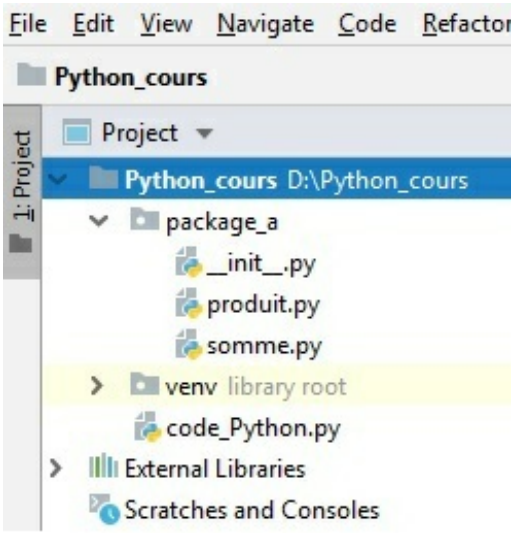
Après la validation, un dossier est ajouté avec le fichier « __init__.py ».



Pour créer un sous-package vous faites un clic droit sur le package voulu et suivez les mêmes étapes précédentes.

Importer des packages

Avant l'importation voilà la structure du projet :



On a le dossier parent (Python_cours) qui contient le fichier script « code_python.py » où le code sera écrit et aussi le package « package_a » qui contient « somme.py, sproduit.py et __init__.py ».

Méthode 1

Cette méthode consiste à utiliser le mot réservé « import » suivi par le chemin complet jusqu'au nom du module que vous voulez l'importer. Cette chemin peut contenir aussi les sous-package si le module existe dans l'un des sous-packages.

Avec cette méthode on a obligé d'écrire chaque fois le chemin complet pour accéder à une fonction, la seule solution c'est renommée le nom d'importation par un nouveau nom.

Exemple :

```
import package_a.somme

x = package_a.somme.sm(5, 2)
print(f"La somme de 5 et 2 égale {x}")
```

Résultat :

```
La somme de 5 et 2 égale 7
```

Exemple :

```
import package_a.somme as som_1

x = som_1.sm(5, 2)
print(f"La somme de 5 et 2 égale {x}")
```

Méthode 2

Cette méthode consiste à utiliser le mot réservé « from » et « import ».

Exemple :

```
from package_a import somme

x = somme.sm(5, 2)
print(f"La somme de 5 et 2 égale {x}")
```

Méthode 3

Cette méthode consiste à utiliser le « joker » * pour importer tous les modules d'un seul coup. Mais ça ne marche pas si vous n'ajoutez pas une instruction dans le fichier « __init__.py » qui dit à Python les noms des sous-package et les modules qui existe dans le package, dans notre exemple on va ajouter la ligne suivante sans le fichier « __init__.py » qui existe dans le package « package_a » :

```
__all__ = ["produit", "somme"]
```

Note : Mette à jour cette liste après la suppression ou l'ajout d'un nouveau module ; par ce que l'importation avec le « joker » * import seulement la liste entrer par l'utilisateur.

Exemple :

```
from package_a import *

x = somme.sm(5, 2)
print(f"La somme de 5 et 2 égale {x}")
y = produit.pr(7, 2)
print(f"Le produit de 7 et 2 égale {y}")
```

Résultat :

```
La somme de 5 et 2 égale 7
Le produit de 7 et 2 égale 14
```

Les exceptions

Ce sont des instructions qui permet de gérer les erreurs lors de l'exécution de votre programme. Ces instruction test la réussite d'une instruction et dans le cas d'erreur vous pouvez mettre votre message d'erreur personnalisé.

Syntaxe :

```
try:
    # Code pouvant générer une exception
except:
    # Code en cas d'exception
else:
    # Code en cas de non exception

# code dans tous les cas
```

Exemple :

```
try:
    x = int("2")
except:
    print("Erreur de conversion")
else:
    print("La conversion est bien faite")
    print(f"La somme égale {5 + x}")
```

Résultat :

```
La conversion est bien faite
La somme égale 7
```

Exemple :

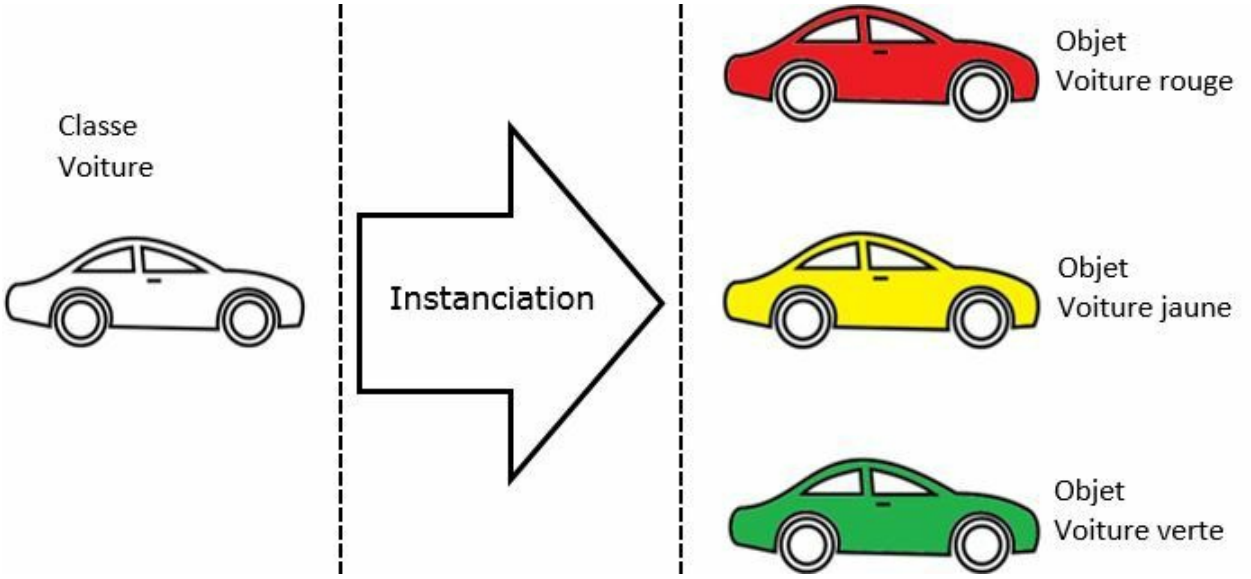
```python
try:
    x = int("deux")
except:
    print("Erreur de conversion")
else:
    print("La conversion est bien faite")
    print(f"La somme égale {5 + x}")
```

Résultat :

```
Erreur de conversion
```

La programmation orientée objet

La programmation orientée objet c'est une manière de codé ou un concept de programmation qui permet d'organisé votre code via une outille qui s'appelle classe, cette dernière est considérée comme un plan de conception des objets qui sont des instances de cette classe contenant des attribues et des méthodes.



Classe

Pour créer une classe on utilise le mot réservé class, cette création peut se faire dans le début de votre script mais vous pouvez mettre votre classe séparé dans un module à importer.

Syntaxe :

```
class nom_classe:
    bloc d'instruction
```

Exemple :

```
class etudiant:
    def __init__(self, nom_etu, age_etu):
        self.nom = nom_etu
        self.age = age_etu

    def affiche_age(self):
        print(f"{self.nom} est âgé de {self.age} ans")

    def modif_age(self, new_age):
        self.age = new_age


etd1 = etudiant("user1", 20)
etd2 = etudiant("user2", 30)
print(f"Le nom de etd1 = {etd1.nom}")
print(f"L'age de etd1 = {etd1.age}")
etd1.modif_age(35)
etd1.affiche_age()
```

- __init__ : c'est le constructeur d'objet, il est appelé automatiquement lorsque vous créez un objet
- Self : représente l'instance de la classe, c'est-à-dire qu'il fait référence à l'objet lui-même
- nom_etu et age_etu : paramètre récupérer lorsque vous créez un objet
- self.nom et self.age : représente les attributs d'une instance ou d'un objet créé, elles sont similaires aux variables
- etd1 = etudiant("user1", 20) : création d'un objet etd1 avec le nom égale user1 et age égale 20
- etd1.nom : accéder à l'attribut nom de l'objet etd1
- def affiche_age(self) : méthode de classe qui affiche l'âge d'un

l’objet

- def modif_age(self, new_age) : modifier la valeur de l’attribut par la nouvelle valeur new_age
- etd1.modif_age(35) :appelle de la méthode modif_age avec la valeur 35 pour l’objet etd1
- etd1.affiche_age() : appelle de la méthode affiche_age pour l’objet etd1
- etd2.age = 35 : modifier directement un attribut

Résultat :

```
Le nom de etd1 = user1
L'age de etd1 = 20
L'etudiant user1 est âgé 35 ans
L'age de etd2 = 40
```

L'héritage de classe

Vous pouvez créer une nouvelle classe qui se base sur une classe existante, de cette façon vous hérité tous les attributs et méthodes de la classe mère.

Syntaxe :

```
class nom_classe(classe mère):
    bloc d'instruction
```

Vous pouvez hérité de plusieurs classes d'un seul coup.

```
class nom_classe(classe1, classe2,…):
    bloc d'instruction
```

Exemple :

```
class etudiant:
    def __init__(self, nom_etu, age_etu):
        self.nom = nom_etu
        self.age = age_etu

    def affiche_age(self):
        print(f"L'etudiant {self.nom} est âgé {self.age} ans")


class groupe(etudiant):
    def __init__(self, nom_etu, age_etu, gr_etu):
        etudiant.__init__(self, nom_etu, age_etu)
        self.gr = gr_etu


et1 = groupe("use1", 20, 111)
print(f"Le nom de et1 est : {et1.nom}")
```

Résultat :

```
Le nom de et1 est : use1
Le groupe de et1 est : 111
```

Comme vous remarquez, pour hériter depuis une classe qui contient un constructeur vous devez appeler le constructeur de la classe mère et aussi le constructeur de la classe fils doit contient les mêmes paramètres de classe mère avec les nouveaux paramètres.

- etudiant.__init__(self, nom_etu, age_etu) : appeler le constructeur de la classe etudiant

----------**Fin**----------